# C++による簡単実習 3次元CG入門

小笠原 祐治 著

森北出版

# C++による簡単実習 3次元CG入門

小笠原　祐治 著

森北出版

C＋＋による簡単実習

# 3次元CG入門

◆小笠原祐治 著

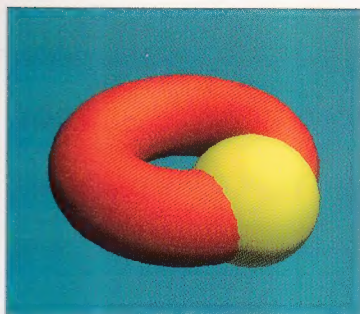
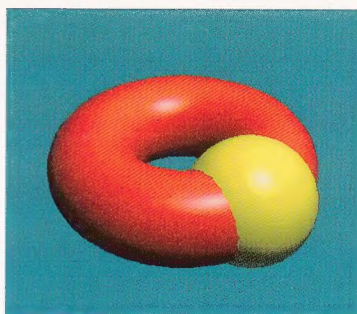
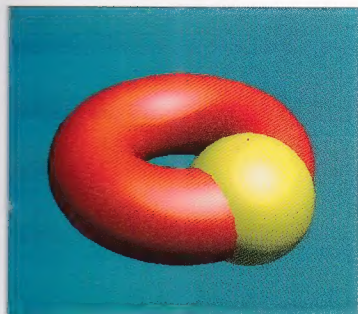
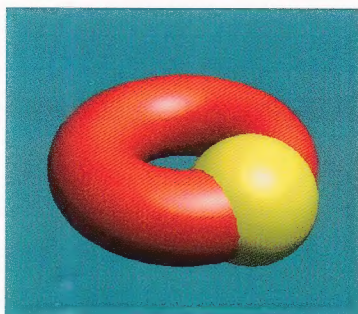
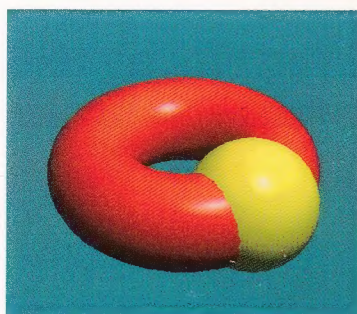
森北出版

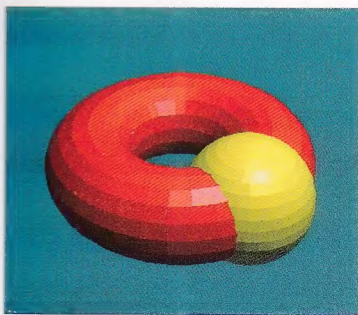
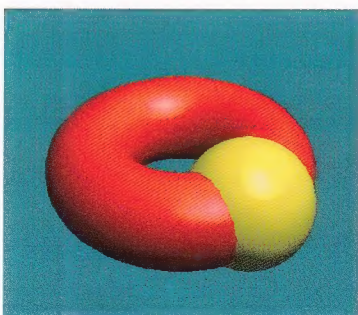
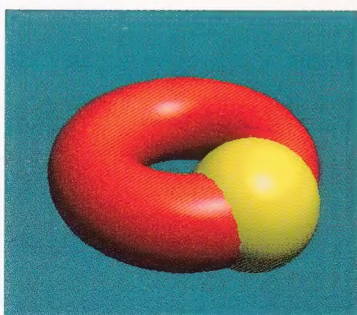
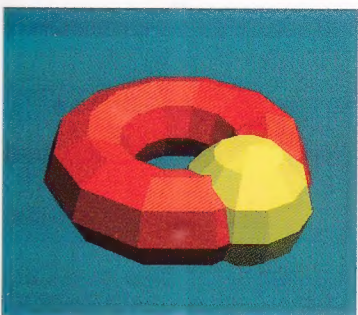
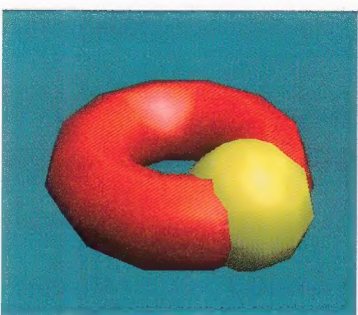
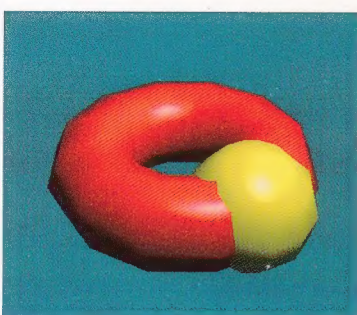
**プログラム例のダウンロードについて**

本書のプログラム例は当社のホームページよりダウンロードできます．詳細は本文11頁をご覧下さい．

(なお，ワークステーション用も考慮中です．)

また，ダウンロード内容はフロッピーディスクでも供給しております．申込方法は本書のカバーの折り返し(内側)をご覧下さい．

◎光学的モデル

口絵 1　ランバーモデル
（演習 6-2）

口絵 2　フォンモデル $n$=20
（演習 6-2）

口絵 3　フォンモデル $n$=5
（演習 6-2）

口絵 4　フォンモデル $n$=10
（演習 6-2）

口絵 5　フォンモデル $n$=40
（演習 6-2）

◎シェーディング

口絵 6　フラットシェーディング
（EX6_1.cpp）

口絵 7　グーローシェーディング
（EX6_2.cpp）

口絵 8　フォンシェーディング
（EX6_3.cpp）

口絵 9　フラットシェーディング
（演習 6-1）

口絵 10　グーローシェーディング
（演習 6-1）

口絵 11　フォンシェーディング
（演習 6-1）

## 口絵の説明（口絵 1〜11）

**◎光学的モデル（6 章）**

・トーラス（ドーナツ形状）と球体が交差している形状を描画している．

・口絵 1（ランバーモデル）と口絵 2（フォンモデル）を比較すると，口絵 2 には光沢があることがわかる．

・口絵 2〜5 を比較すると，光沢が異なる．

**◎シェーディング（6 章）**

・口絵 6 は，平面で形状を Z バッファ法で描画している．

・口絵 7 と口絵 8 はグーローシェーディングおよびフォンシェーディングにより，曲面のように描画している．光沢に注意すると，口絵 8 の方がより自然であることがわかる．

・口絵 9〜11 は，口絵 6〜8 のポリゴン数を減らした場合の描画である．
口絵 6〜8 では，トーラスが 18×36 ポリゴン，球体が 18×36 ポリゴンである．
口絵 9〜11 では，トーラスが 8×12 ポリゴン，球体が 18×36 ポリゴンである．

←

## 口絵の説明（口絵 12〜20）

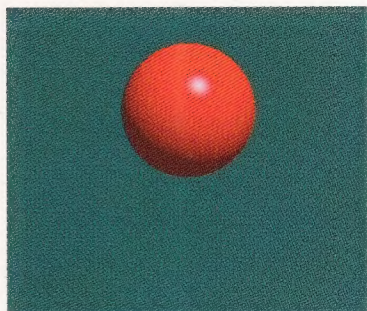
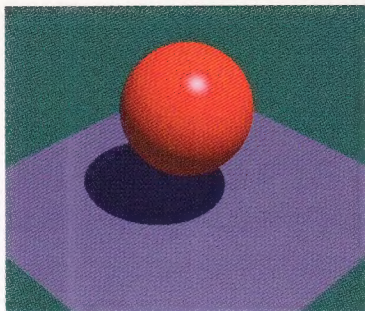
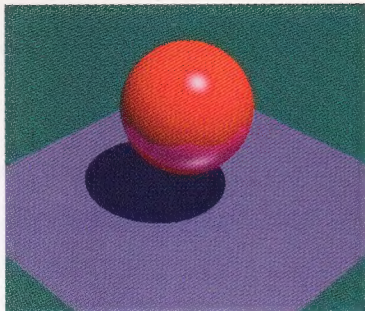
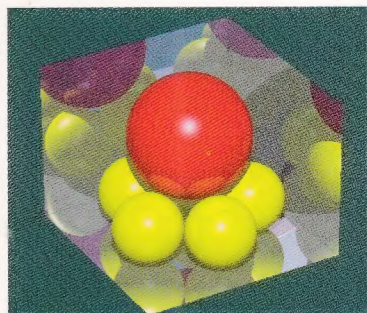
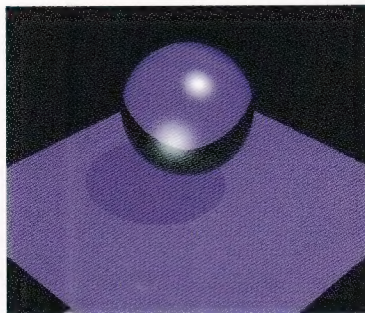
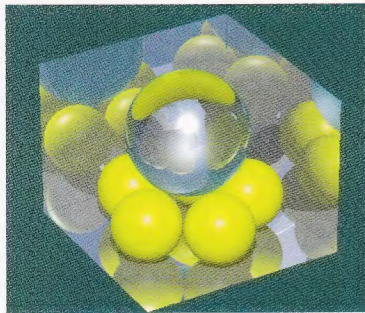
◎**レイトレーシング（7章）**

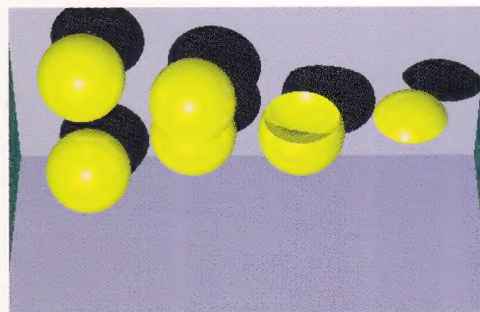
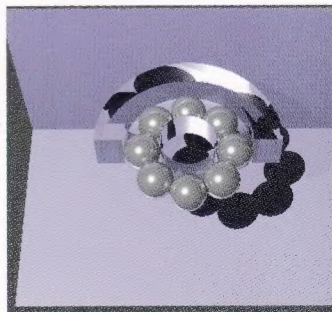
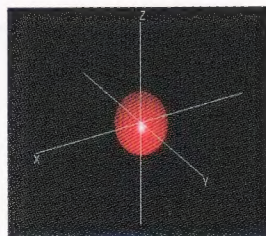
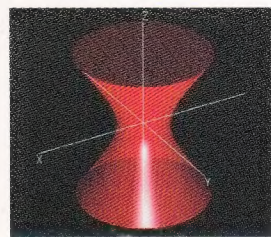
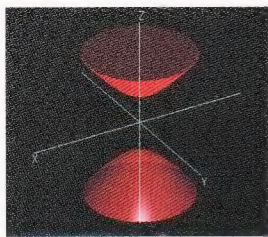
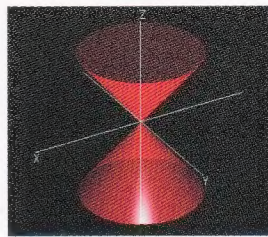
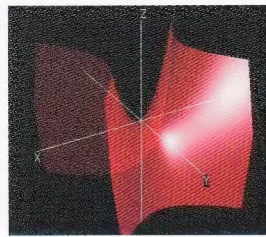
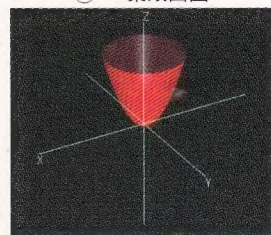
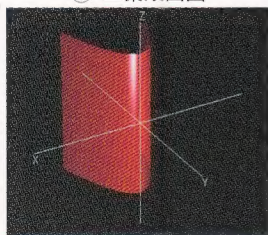
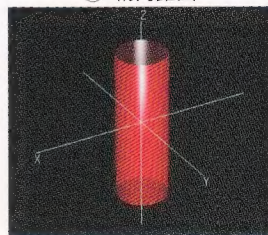
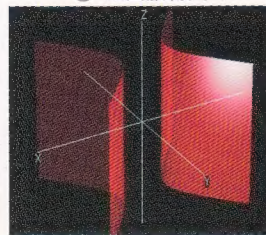
- 口絵 12 は，球体を描画している．
- 口絵 13 は，床に球体による影があるところを描画している．
- 口絵 14 は，さらに球体に床が映り込んでいることがわかる．
- 口絵 15 は，球体の 2 段積ねが壁や床に映り込んでいる．さらに，球体にも壁，床，ほかの球体が映り込んでいることがわかる．
- 口絵 16 は，球体が透明なので，床の端が屈折して見えていることがわかる．
- 口絵 17 は口絵 15 の赤い球体を透明にしたもので，壁に反射した球体が屈折して見えていることがわかる．

◎**CSG モデル（7章）**

- 口絵 18 は，球体の演算で和，差，積を描画している．
- 口絵 19 は，ベアリングのカットモデルを描画している．
- 口絵 20 は，CSG モデルで比較的容易に使用できる 2 次曲面の例を示す．

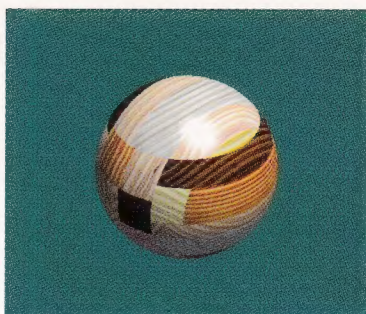
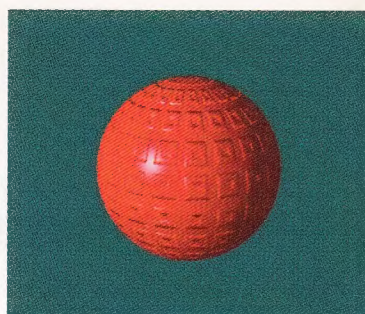
→

◎レイトレーシング

口絵 12　球体（EX7_1.cpp）

口絵 13　影を付ける（EX7_2.cpp）

口絵 14　映り込み（EX7_3.cpp）

口絵 15　球体の 2 段積み（AP3_2.cpp）

口絵 16　屈折（EX7_4.cpp）

口絵 17　球体の 2 段積み（AP3_3.cpp）

◎CSGモデル

口絵 18　立体の演算（和，差，積）（EX7_5.cpp）

口絵 19　ベアリング（EX7_6.cpp）

① 楕円面

② 一葉双曲面

③ 二葉双曲面

④ 楕円錐面

⑤ 双曲放物面

⑥ 楕円放物面

⑦ 方物柱面

⑧ 楕円柱面

⑨ 双曲柱面

口絵 20　2 次曲面（AP3_1.cpp）

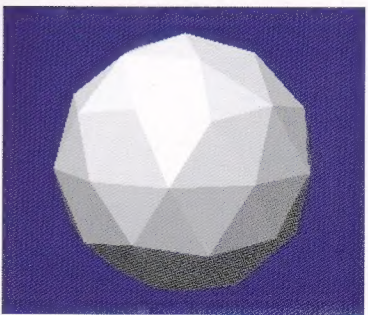
◎マッピング

口絵 21　テクスチャマッピング
（EX8_1.cpp）

口絵 22　ソリッドテクスチャリング
（EX8_2.cpp）

口絵 23　バンプマッピング
（EX8_3.cpp）

口絵 24　サッカーボール
（演習 8-1）

口絵 25　リフレクションマッピング
（EX8_5.cpp）

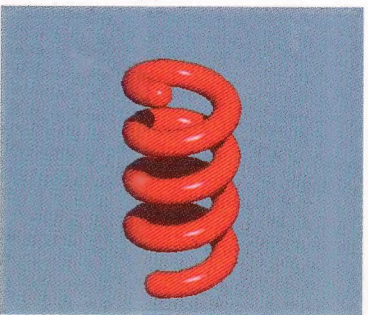
口絵 26　リフラクションマッピング
（EX8_5.cpp）

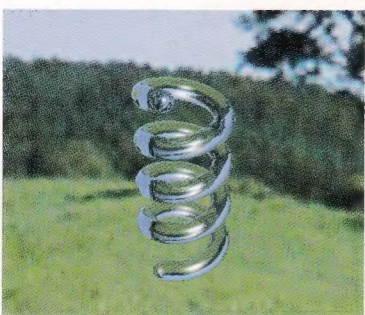
◎アンチエイリアシング

口絵 27　多面球体
（AP2_3.cpp）

口絵 28　アンチエイリアシング
（AP2_3.cpp）

◎メタボール

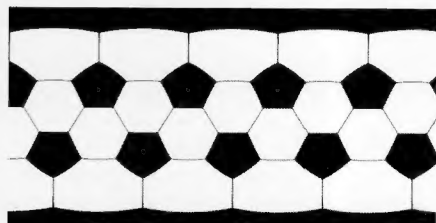
口絵 29　メタボール

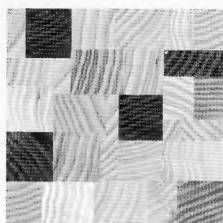
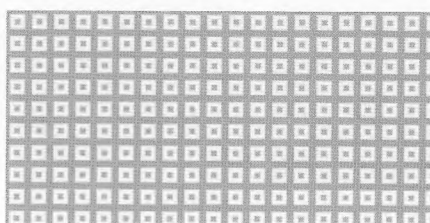
◎作品 1 「スプリング」

口絵 30　ポリゴンモデル
（AP2_1.cpp）

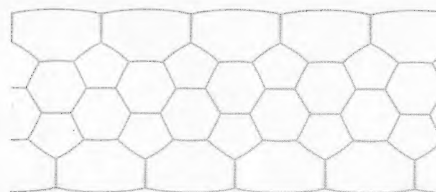
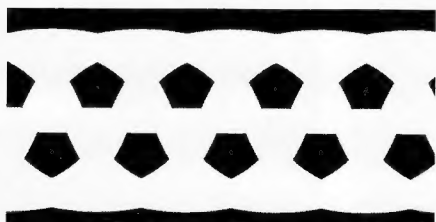
口絵 31　鏡面物体（レイトレーシング）
（AP4_1.cpp）

口絵 32　透明物体（レイトレーシング）
（AP4_1.cpp）

## 口絵の説明（口絵 21～32）

### ◎マッピング（8 章）

・口絵 21 は，球体の表面に下図の画像を貼り付けている．

・口絵 22 は，下図の画像を用いて寄せ木細工のような球体表現している．

・口絵 23 は，球体の表面に下図の画像を用いて凹凸を付けている．

（口絵 22 で使用）

（口絵 23 で使用）

・口絵 24 は，球体の表面に下図の画像を用いて模様と凹凸を付けている．

・口絵 25 は，背景が鏡のような球体に映り込んでいる様子を表現している．

・口絵 26 は，背景が透明な球体により屈折して見える様子を表現している．

### ◎アンチエイリアシング（7 章）

・口絵 27 は，多面球体である．

・口絵 28 は，アンチエイリアシング処理をしており，境界のジャギー（ギザギザ）が軽減されている．

### ◎メタボール（7 章）

・口絵 29 は，2 つのメタボールを近づけた場合の描画である．

### ◎作品 1 スプリング（付録 4）

・口絵 30～32 は，形状はスプリングであり，材質は光沢があり，鏡面，透明としている．口絵 31，32 は，*B* スプライン曲面（パラメトリック曲面）を用いている．

## 口絵の説明（口絵 33～44）

**◎作品 2　コーヒーカップ（付録 4）**

・口絵 33～38 は，各種描画方法で描画している．
　口絵 37，38 は，*B* スプライン曲面（パラメトリック曲面）を用いている．

**◎作品 3　多面球体（付録 4）**

・口絵 39～41 は，球体の 3 角形を組み合わせた立体で近似している．

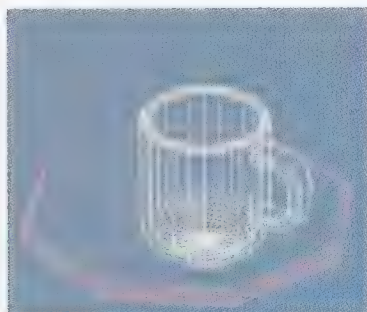
**◎作品 4　レンズ（付録 4）**

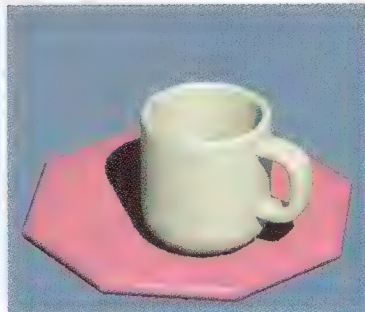
・口絵 42，43 は，CSG モデルでレンズを描画している．
　凸レンズの場合には背景が反転し，凹レンズの場合には背景が小さく見える．

**◎作品 5　テーブルと灰皿（付録 4）**

・口絵 44 は，CSG モデルでテーブルと灰皿を描画している．

→

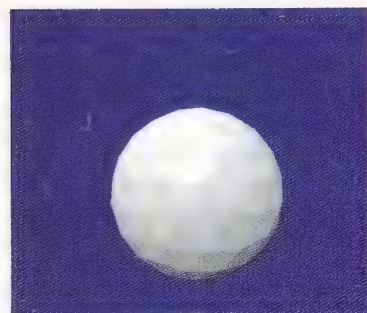
◎作品 2 「コーヒーカップ」

口絵 33　ワイヤーフレーム（AP2_2.cpp）

口絵 34　フラットシェーディング（AP2_2.cpp）

口絵 35　グーローシェーディング（AP2_2.cpp）

口絵 36　フォンシェーディング（AP2_2.cpp）

口絵 37　鏡面物体（レイトレーシング）（AP4_2.cpp）

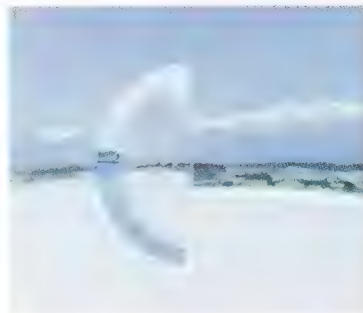
口絵 38　透明物体（レイトレーシング）（AP4_2.cpp）

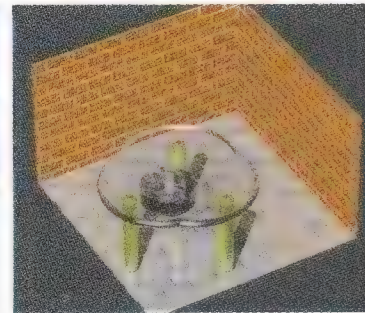
◎作品 3 「多面球体」

口絵 39　ポリゴンモデル（AP2_3.cpp）

口絵 40　鏡面物体（レイトレーシング）（AP4_3.cpp）

口絵 41　透明物体（レイトレーシング）（AP4_3.cpp）

◎作品 4 「レンズ」

口絵 42　凸レンズ（AP3_4.cpp）

口絵 43　凹レンズ（AP3_4.cpp）

◎作品 5 「テーブルと灰皿」

口絵 44　テーブルと灰皿（AP3_5.cpp）

# 推薦文

本書は，“ＣＧのプログラミング”を学びたい方，“プログラミングを楽しく”学びたい方にお薦めします．また，「ＣＧ検定１級」の受験者には，特にお薦めします．

目次で示される３次元ＣＧの基本的な技法すべてについて，その理論の説明のみならず，プログラム例が示されています．口絵のカラー画像が著者の努力を物語っています．口絵２９を除きすべてこの掲載プログラムで生成されたＣＧです．これまでの３次元ＣＧ技法を扱った類書との違いがよくわかると思います．

理論からプログラムまでは結構距離があります．例えば，レイトレーシングのプログラムを書こうとすると，直線や平面の方程式の求め方，それらの交点の求め方，交点の多角形面に関する内外判定のしかた，ベクトルの内積や外積の求め方，正反射方向のベクトルの求め方，などなど，たくさんの具体的な計算法を知らなければいけません．また，解説をする方からいえば，技法の解説のみならず，プログラムも示すためには結構体力がいります．本書では，通常は簡単な説明で済まされがちな環境マッピングについてもきちんとプログラムを含めて解説されています．説明図も，手を抜くことなく，実に丁寧に書かれています．

本書を読んでいると，オペレーティングシステムやコンパイラの原理を理解しようとして，その機能の概念的な説明や，基礎となる手法の解説書をたくさん読むより，小さくとも完備されたプログラムの掲載された本を読むことが非常に効果的であったことを思い出します．本書は，ＣＧに関するそのような良書であると思います．

1999 年 1 月

千葉則茂　岩手大学

# はじめに

コンピュータグラフィックス（CG）は，CAD/CAMはもとより映画，コマーシャル，ゲームにまで利用されるようになり，ますます身近なものとなってきています．近年，コンピュータの性能向上により，パソコンでもフルカラー（約1,600万色）の３次元CGができるようになり，見るだけでなく自分でCG画像を作成できるようになってきました．

CG関係の書籍には，CGのアルゴリズムを学ぶためのものと，CGソフトの使用法を学ぶためのものがあります．本書は前者を目指したものです．前者に属するものには，理論やアルゴリズムの解説を中心に行うものと，実際のプログラムも示した実用的なものがあります．本書は後者に属するものです．

本書の特長としては，利用範囲が広いようにパソコンを使用して，プログラム例を介して容易に３次元CGの実習ができるようにしていることです．一般にWindowsのプログラムは，イベント駆動型にする必要があります．しかし，本書ではイベント駆動型にせずに（MS-DOSのように），プログラムを記述することができるようにしています．また，点や線分などを数学の教科書のように記述することができます．このように，本書のプログラム例では，CGのアルゴリズムを主に記述しており，アルゴリズムに直接関係のない部分はヘッダファイルに記述することでできる限り簡略化して，簡潔でわかりやすいプログラミングを心掛けています．

上記を実現するために，プログラミング言語としてC++言語（Visual C++またはC++ Builder, Borland C++）を使用していますが，読みやすいように簡単な機能だけを使用しています．また，本書でもC++言語について簡単に説明していますので，C言語を学習済みであれば特にC++言語の書籍を用意する必要はありません．

以下に詳細を示します．

（１）　テキストの利用範囲が広い

・大学，短大の情報系の学科およびコンピュータ専門学校で，CGアルゴリズムを学ぶ学生を対象としている．

・学校で使用しやすいように，15回程度（90分/回）の授業で扱える内容としている．
　１～２章（１～２回），３章（３回），４章（１回），５章（２回），
　６章（２回），７章（３回），８章（２回）

・広く普及しているWindows（95以上/NT3.51以上）パソコンで実習ができる．

（２）　CGのアルゴリズム（原理）をわかりやすく記述

・できるだけ，アルゴリズムを前面に出している．

・プログラム例を豊富に載せている．

・(3)および(4)により，簡潔でわかりやすいプログラムを目指している．

（3）　ウィンドウシステムの知識が不要

・ウィンドウシステムを簡単に使用するために単純な関数群（12関数）を用意している．

・再描画を考慮する必要がない（再描画するためのプロセスを用意している）．

・簡単に複数ウィンドウを使用できる．

（4）　プログラムの記述が簡潔（数学の教科書に近い記述が可能）

・*xyz* の各座標ごとの記述は必要ない（プログラムがすっきりする）．

【例】点 b (1,1,1) を変換マトリックス $t$ で，移動する．点 a に移動．

```
Point  b=Point(1,1,1);
Point  a=t*b;      座標変換の計算
w.point(a);        点aの表示
```

・引数の個数が可変である（必要な個数だけ記述する）．

【例】点 (0,0,0)，点 (1,0,0)，点 (1,1,0) を結ぶ複数線分 $l$．

```
Line  l=Line(Point(0,0,0),Point(1,0,0),Point(1,1,0));
w.line(l);         複数線分lの表示
```

本書のプログラム例（ヘッダファイル，画像ファイル，プロジェクトファイルを含む）は，インターネットを介してダウンロードすることができます（p.11参照）．ダウンロードするファイルには，プロジェクトファイルが含まれているので，すぐにプログラム例を実行できます．しかし，実習を行うに当たっては，理解を深めるためにCGのアルゴリムを記述しているプログラム例は入力して，ヘッダファイルなどダウンロードしたものを使用することをお勧めします．

本書の執筆にあたり，御助言いただいた岩手大学の千葉則茂教授に感謝いたします．

また，出版のために御尽力いただいた森北出版の田中節男氏に感謝いたします．

1999 年 1 月

みちのく盛岡にて　小笠原祐治

# 目　次

## 付　録　99



## 演習問題の解答例　126

## 巻末リスト　135

## さくいん　151

# 1．準　備

近年，グラフィックス関連の著書としては，C言語を用いたものがよく見受けられます．本書では，C++言語を用いることによって，グラフィックス・アルゴリズムを簡潔でわかりやすくプログラミングすることを目指しています．C++言語といっても，簡単な機能だけを使用しているので，本書のプログラム例は容易に読むことができます（特にC++言語の書籍を用意する必要はありません）．

また，ウィンドウシステムを簡単に使用することができるようにするために，単純な関数群（WinクラスとColorクラス）を用意しており，ウィンドウシステムの使用方法をあまり習得しなくても済むようにしています．

本章では，掲載しているプログラム例の実行環境，およびC++言語の特長(C言語と異なる部分)について説明しています．ウィンドウの操作については２章で説明します．

## 1．1　プログラム例の実行環境

本書では，プログラム例を多数記載しています．プログラム例はインターネットでダウンロードすることができ，パソコン上で実行できます．プログラム例の実行条件は，以下の通りです．

① Visual C++4.0以上，またはC++ Builder, Borland C++5.0J以上が動作可能であること．

- OSがWindows95以上，またはWindowsNT3.51以上であること．
  ( Windows95/98/NT4.0で確認済み)．
- メモリが16MB以上であること．

② 発色数が32, 768色以上であること．

③ ハードディスクに35MB以上の空きがあること．

一般にWindowsでのプログラムでは，ウィンドウの再描画を考える（イベント駆動型にする）必要がありますが，描画用（再描画含む）のプロセスを用意しており，プログラムで意識しなくても自動的に再描画されます．これにより，プログラミングをするときに，MS-DOSのプログラムのように記述できます．

なお，C言語ユーザが容易に読めるようにするため，C++言語の機能をあまり多用しないようにしています．また，main関数およびクラスの記述を簡潔にし，理解しやすさを目指しています．

## 1．2　C++言語の使用

C++言語を使用した場合，C言語を用いた場合に対して以下に示す長所があります．

（1）　座標変換などの演算を，簡潔に記述できる．

簡単にいうと，「行列やベクトルなどの計算を，数学の教科書での表現に近い形で記述できる」ことです．C++言語では，演算子の多重定義（オーバーロード）という機能があり，クラ

ス（C言語の構造体を拡張したもの）間の演算（例えば，＋，－，＊，／）を定義することができます．この機能を利用することによって，座標変換の計算（行列のかけ算）を下記のように記述することができます．

```
a=t*b;
```

$t$ は変換行列，$b$ は変換前の座標，$a$ は変換後の座標を表します．あらかじめ演算の定義が必要ですが，非常に直感的で簡潔な記述ができます．C言語では，$x,y,z$ の各座標の計算をそのつど記述するか，あるいは演算子に相当する関数（例えば，multiという関数）を用意しておき，その関数を用いて行う必要があります．

（2）　関数をラフに使用できる．

・C言語では，関数の引数の型あるいは引数の数が異なれば，関数名を変える必要があります．C++言語では，同じ関数名でも許されます．何をするかという関数名を覚えていればよく，引数の型はあまり気にせずに使用できるようにすることが可能です．

・C++言語では関数の引数の省略値の設定が可能であり，次のような場合には大変便利です．

```
void func( float a, float b=0., float c=0., float d=0. );    ﾌﾟﾛﾄﾀｲﾌﾟ宣言
func( 2., 1. );                                              ①
func( 2., 1., 0., 0. );                                      ②
```

上記のように，プロトタイプ宣言で省略値を記述しておきます．①のような記述は引数の $c,d$ が省略されていますので，②とみなされ引数 $c,d$ は0となります．本書では，多角形の記述の際に，頂点の座標を指定する場合に使用しています（頂点の個数を指定なくても済みます）．

## 1.3　C++言語について

本書のプログラム例で使用するC++言語の機能について，簡単に説明します．

### 1.3.1　関数のオーバーロード（多重定義）

C++言語では，名前が同じ関数を複数定義することができます．ただし，各関数は引数の型あるいは引数の個数が異なる必要があります．その例を示します．

```
int       min(int x, int y);                     ①
float     min(float x, float y);                 ②
float     min(float x, float y, flaot z);        ③
```

呼び出される場合の引数の個数と型によって，どの関数が使用されるかが決まります．

```
float     a,b,c;
a=min(1, 2);
b=min(1.,2.);
c=min(1.,2.,3.);
```

上記の例では，変数 $a$ への代入では①の関数が使用され，変数 $b$ では②，変数 $c$ では③の関数が使用されます．

### 1.3.2　引数に関するC言語とC++言語の違い

関数の引数におけるC言語とC++言語の違いを見ていきます．

1)　デフォルト引数

C++言語では，引数の省略値を定義することができます．その例を示します．

```
float        min(float x, float y, float z=0.0);
```

次に示すように，引数が2つで呼び出された場合は，3つ目の引数は0.0とみなされます．

```
a=min( 1.0, 12.0);
```

2)　引数の型変換

(1)のmin関数が定義されている場合，次のようにint型の引数$i$で呼び出すと，引数がfloat型に自動的に変換され，min関数が使用されます．

```
int  i=4;
a=min( i, 12.0);
```

3)　参照型の引数

次のように関数を定義すると，引数を参照渡し（pass by reference）にすることができます．

```
void   exchange(int& x, int& y){    int a=x;    x=y;    y=a;       }
```

次のように呼び出すと，int型の変数$e$と$f$間で値が入れ替わります．

```
exchange( e, f);
```

### 1.3.3　一時変数

C++言語では「変数の宣言は実行文の前で行う」という制約はありません．プログラムのどこで変数を宣言してもよく，その変数は宣言を行ったブロック内（"{" と "}" で囲まれた範囲）で有効です．次に例を示します．

```
main()
{
        float    c,d,e;
        ～
        ～
        for(int i=0; i<10; i++){                                        ①
                float    a=c*d;                                         ②
                e=cos(a);
        }                                                               ③
        ～
        ～
}
```

①と②で変数を宣言しています．①ではfor文の中で変数$i$を宣言しており，main関数内で有効です．また，②では変数$a$を宣言しており，③までの間で有効です．

### 1.3.4 クラス（コンストラクタ，メンバ関数）

クラス（class）はC言語の構造体を拡張したものと考えることができます．

構造体はデータの構造を定義したものでしたが，クラスはデータの構造に加えてその処理（メンバ関数）まで定義したものです．

例として，Pointクラスについて見ます．このクラスは点の座標を表しており，$x,y,z$ のfloat型データ（データメンバ）およびコンストラクタ（①～②）からなっています．

```
class Point{
        public:
        float   x,y,z;
        Point()                     {  x=0.;  y=0.;  z=0.;  }          ①
        Point(float ix, float iy, float iz=0.)
                                    {  x=ix;  y=iy;  z=iz;  }          ②
};
```

コンストラクタは，クラス型の変数が宣言された場合の初期化処理を記述したものです．上記の例の場合，コンストラクタが複数定義されていますが，引数の型および引数の数によって使い分けられます．次のように，Point型の変数が宣言された場合，引数がないので①のコンストラクタが使用され，$x,y,z$ はそれぞれ0.0に初期化されます．

```
Point  p;
```

一方，次のように宣言された場合，引数が2つあるので②のコンストラクタが使用され，$x=5.0, y=10.0, z=0.0$ と初期化されます．

```
Point  p(5,10);
```

なお，コンストラクタの定義では，関数の型は指定できません．また，コンストラクタは必須ではなく，省略してもかまいません．

データメンバへのアクセスは，構造体と同ように行うことができます．Point型の変数名 $p$ の後に"."を付けて，メンバ名を記述します．

```
Point  p(5,10);
float  a=p.x;                                aには5.0が代入される
```

次に，メンバ関数の使用方法について説明します．描画を行うためのWin2Dクラスを用いて，ウィンドウを開いて線分を描画する例を以下に示します．Win2Dクラスの記述があり，その後にそれを使用して線分の描画を行うmain関数があります．Win2Dクラスの記述の中の②，③，④がメンバ関数であり，その記述方法は２種類あります．②はメンバ関数の宣言と同時に処理も記述しています．③，④はメンバ関数の宣言であり，その処理は⑤，⑥で記述しています（３章のプログラム例で使用しているWin2Dクラスの記述とは多少異なります）．

```
//      ウィンドウ用のクラス
class  Win2D
{
        #define  HEIGHT_DEF       350
        #define  WIDTH_DEF        400
        float    org_x,  org_y;            //  原点の位置（左上が 0,0 ）
```

```
        Win      w;
        public:
        Win2D(char* wn="Win2D", int x=WIDTH_DEF, int y=HEIGHT_DEF,
                                         Color c=WHITE);                      ①
        void     point(Point p)    {  w.point(org_x+p.x,org_y-p.y);   }      ②
        void     move (Point p);                                              ③
        void     line (Point p);                                              ④
        ~
};
void   Win2D::move (Point p)     {  w.move (org_x+p.x,org_y-p.y);   }      ⑤
void   Win2D::line (Point p)     {  w.line (org_x+p.x,org_y-p.y);   }      ⑥

main()
{
        Win2D    w1;                                                          ⑦
        Point    a=Point(0,0), b=Point(10,10);
        w1.move(a);                                                           ⑧
        w1.line(b);                                                           ⑨
        ~
```

mainプログラムの処理は，⑦の記述で①のコンストラクタが呼び出され，ウィンドウが開きます．⑧,⑨でWin2Dクラスのメンバ関数を使用して，座標(0,0)から座標(10,10)へ線分を描画します．データメンバと同じように，Win2D型の変数名w1の後に"."を付けて，メンバ関数名および引数を記述します．

### 1.3.5　new,delete演算子

new演算子は記憶領域（ヒープ領域）の割り当てを，delete演算子は記憶領域の解放を行うものです．以下にfloat型の配列の例を示します．配列のサイズを変数で指定できることに，注意して下さい（プログラミングにあまり立ち入らない場合には，次節へ進んで下さい）．

```
    int  n=10;
    float*       a=new float[n];                                   ①
    for(int i=0; i<10; i++)a[i]=1;
      ~
      ~
      ~
    delete[] a;                                                    ②
```

①の"new"でfloat型配列の領域を確保しています．"float"の代わりにクラス(例えば"Point")を指定することもできます（C++言語でのnewはC言語の"malloc"などに相当します）また，②の"delete"は，newで確保した領域が不要になった場合，それを解放するものです（C++言語でのdeleteはC言語の"free"などに相当します）．

newで確保した領域は，そのブロック（"{"と"}"で囲まれた範囲）から出ても自動的に解放されるわけではないので，deleteで解放する必要があります．

### 1.3.6　演算子のオーバーロード（多重定義）

C++言語の場合，クラスの演算処理を定義することができます．Point型間の加算"+"の演算子の定義の例を示します．

```
Point  operator+(Point& a,Point& b)  {  return( Point(a.x+b.x,a.y+b.y) );  }
```

次に示すような場合，上記の演算子の定義が使用されます．

```
Point  p1(5,10);
Point  p2(10,10);
Point  d=p1+p2;
```

下記のように，同様な演算を記述することもできます．

```
Point  d=Point(5,10)+Point(10,10);
```

演算子の定義は，引数の片方がクラスであれば可能です．

実数(float型)とPoint型の掛算"＊"の例を示します．

```
Point  operator*(float a,Point& b)      {return( Point(a*b.x,a*b.y) );     }
```

次に示すような場合，上記の演算子定義が使用されます．

```
Point  p1(5,10);
Point  p2=2.5*p1;
```

# 2．基本グラフィック関数

この章では，本書で使用するウィンドウ操作関数と描画関数，およびプログラム例の実行方法について説明します．３章以降のプログラム例では，最終的に本章で説明する関数を使用して，ウィンドウ操作および描画を行っています．

## 2.1 座標系とスキャン変換

一般のコンピュータで使用されている表示装置（ＣＲＴ，液晶など）は，ラスタスキャン型と呼ばれる表示装置です．そのため，図2-1に示すように表示装置のスクリーン（画面）に画素（pixel）が格子状に配列されており，その画素の色を変更することによって描画が行われます．このように，各画素の色を変更することによって，画像は描かれます．

図2-1　スクリーンと画素の関係

スクリーン内の各画素は，その位置に応じた座標値によって識別されます．その座標をスクリーン座標ということにします．スクリーンの横方向の座標を*X*座標，縦方向を*Y*座標とします．その座標値を図2-1に示します．左上が，座標(0,0)となります．

図2-2　線分の描画

例えば，直線を描画する場合について考えてみましょう．座標(11,11)から(24,16)への線分は，図2-2に示すように複数の画素で描画されます．始点を座標（$x_0, y_0$），終点を座標（$x_1, y_1$）とすると，線分を構成する各画素の座標$(x,y)$は式(2-1)で示されます．ここで，始点$(x_0, y_0)=(11,11)$，終点$(x_1, y_1)=(24,16)$であるので，$X$座標の値$x$は11から24までであり，$Y$座標の値$y$は式(2-1)により求められます．式（2-1）の値は実数値となりますが，四捨五入した整数値を用います．

$$y=\frac{y_1-y_0}{x_1-x_0}\cdot(x-x_0)+y_0 \qquad (2\text{-}1)$$
$$x=x_0, x_0+1, x_0+2, \cdots, x_1$$

図2-2の例では，始点（$x_0, y_0$）と終点（$x_1, y_1$）間で変位が大きい方の座標（$X$座標の変位は17，$Y$座標の変位は５）は$X$座標であるので，$X$座標を始点から終点まで１ずつ変化させました．$Y$座標の方が変位が大きい場合は，$Y$座標と$X$座標を入れ替えて画素座標を求めます（式(2-2)を使用）．このように，線分などを複数の画素で近似することをスキャン変換といいます．

$$x=\frac{x_1-x_0}{y_1-y_0}\cdot(y-y_0)+x_0 \qquad (2\text{-}2)$$
$$y=y_0, y_0+1, y_0+2, \cdots, y_1$$

## 2.2　ColorクラスとWinクラス

画素の色は，赤，緑，青（３原色）の各成分の明るさで指定することができます．色を記述するために，Colorクラスを用意しています（Colorクラスの記述は，win.hファイルにあます．リスト2-1に示すように，３原色（赤，緑，青）の各明るさの値（0.0～1.0）で色を表しており，(0,0,0)で黒，(1,1,1)で白となります．描画できる色数は無限ではなく，ハードウェアによって決まっているので，最も近い色で描画されます．Colorクラスは，以下のような使い方ができます．

・３原色の設定

```
Color          c=Color(0.5, 0.5, 0.5);
```

**変数cに灰色（赤=0.5, 緑=0.5, 青=0.5）が代入される．**

・色の加算

```
Color          a=Color(0.0, 0.3, 0.5);
Color          b=Color(0.5, 0.5, 0.0);
Color          c=a+b;
```

**変数cに色（赤=0.5, 緑=0.8, 青=0.5）が代入される．**

・色のかけ算（明るさを変える）

```
Color          a=Color(1.0, 1.0, 1.0);
Color          c=0.5*a;
```

**変数cに色（赤=0.5, 緑=0.5, 青=0.5）が代入される．**

あらかじめ定義している描画色を，以下に示します．

```
#define    WHITE    Color( 1.0, 1.0, 1.0 )    // 白
#define    YELLOW   Color( 1.0, 1.0, 0.0 )    // 黄色
#define    MAGENTA  Color( 1.0, 0.0, 1.0 )    // 紫
#define    CYAN     Color( 0.0, 1.0, 1.0 )    // 水色
#define    RED      Color( 1.0, 0.0, 0.0 )    // 赤
#define    GREEN    Color( 0.0, 1.0, 0.0 )    // 緑
#define    BLUE     Color( 0.0, 0.0, 1.0 )    // 青
#define    BLACK    Color( 0.0, 0.0, 0.0 )    // 黒
```

定義済み描画色

ウィンドウ操作や描画に使用するグラフィック関数は，Winクラスに記述されています．そこで定義されているメンバ関数は基本的なものであり，12の関数があります．これらの関数は，スクリーン座標を用いています．そのため，描画を行う際に座標変換など(3章で説明)を行い，最終的にWinクラスの関数によって，スクリーン座標で描画を行います．Winクラスはwin.hファイル(リスト2-1)に記述されいます．Winクラスのメンバ関数の説明を以下に示します．

- void open(char* wn, int x, int y, Color c);
  グラフィックウィンドウ（ウィンドウ名がwn）を開く．（初期設定を行う）x,yはウィンドウのサイズであり，cはウィンドウ内の背景色を指定する．
- void color(Color c);
  描画色を設定する．
- void point(int x, int y);
  座標(x,y)に点を描画する．
- Color pixel(int x, int y);
  座標(x,y)の色を返す．
- void move(int x, int y);
  グラフィックカーソルを座標(x,y)に移動する．
- void line(int x, int y);
  グラフィックカーソル座標から座標(x,y)まで線分を描画する．
- void line(int a[], int n);
  以下の連続したn-1の線分を描画する．
  座標(a[0],a[1])から座標(a[2],a[3])，座標(a[2],a[3])から座標(a[4],a[5])座標(a[4],a[5])から…座標(a[2n-4],a[2n-3])から座標(a[2n-2],a[2n-1])まで．
- void paint(int a[], int n);
  多角形の内部を塗りつぶす．
  多角形の頂点は座標(a[0],a[1])から座標(a[2n-4],a[2n-3])までであり，始点の座標(a[0],a[1])と終点の座標(a[2n-2],a[2n-1])は等しい必要がある．
- int gprintf(char *format,...);
  グラフィックカーソル座標から文字列を描画する．引数の形式は，printf関数と同じ．
- void read(char* fn);
  bmp形式のファイル(ファイル名がfn)を読み込み，ウィンドウを開いて表示する．
- void clear();
  スクリーン内部を消去する．
- void close();
  グラフィックウィンドウを閉じる．

三角形と線分の簡単な図形を描画するプログラム例(EX2_1.cpp)をリスト2-2に，実行結果を図2-3に示します．リスト中のpause関数は入力を待つ関数で，実行すると"PAUSE"という名前のウィンドウが表示され，"OK"ボタンをクリックすることによって終了します．プログラム実では，プログラムを終了してよいかどうかを確認するために用いています．なお，プログラムの実行方法は，2.4節を参照して下さい．

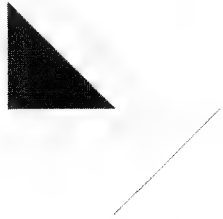
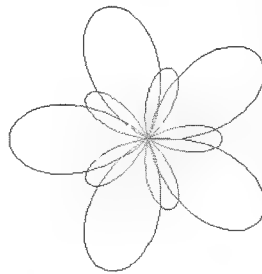



**図 2-3**　実行結果（EX2_1.cpp）　　**図 2-4**　実行結果（EX2_2.cpp）

花形の簡単な図形を描画するプログラム例(EX2_2.cpp)をリスト2-3に，実行結果を図2-4示します．

> **【注意】** 一般的にWindowsのプログラムはイベント駆動型にして，再描画を行わなければなりません．しかし，Winクラスを使用する(win.libをリンクする)ことによって，再描画を考慮しなくても済むようにしています．
>
> また，Windowsでは，UNIXのようなコンソール・ウィンドウを持たないのでprintf, scanf関数が使用できません．そのため，疑似的にprintf, scanf関数を使用できるようにする方法を提供しています．付録を参照して下さい．

## 2.3 プログラム例のダウンロード

本書のプログラム例をダウンロードする手順について説明します.

(1) インターネットで森北出版のホームページに接続し，本書のダウンロードページに行く.
（URLは大文字と小文字を区別しますので，下記の通り入力してください.）

| URL　http://www.morikita.co.jp/soft/3DCG/ |
|---|

(2) 以下の圧縮ファイルを，ダウンロードする.
Visual C++またはC++ Builder，Borland C++がインストールされているドライブのルートディレクトリに，ダウンロードすることを推奨します.

CG_VC4.exe（Visual C++ 4.0の場合）
CG_VC5.exe（Visual C++ 5.0の場合）
CG_VC6.exe（Visual C++ 6.0の場合）
CG_BC5.exe（Borland C++ 5.0の場合）
CG_CB.exe　（C++ Builderの場合）
CG_CB3.exe（C++ Builder 3の場合）

> **【注意】** Visual C++において，ダウンロードしたディレクトリがルートディレクトリでない場合，ソースファイルをプロジェクトに追加し直す必要があります.
>
> Borland C++において，ダウンロードしたドライブにBorland C++がインストールされていない場合，Borland C++のインクルードファイルやライブラリファイルのパス設定を変更する必要があります（オプション->プロジェクト->ディレクトリで変更）. Borland C++がインストールされているディレクトリが，標準と異なる場合にも，同様にパス設定の変更が必要です.
>
> C++ Builder2を使用する場合には，C++ Builder用のファイルを使用して下さい.

(3) CG_VC4.exe，またはCG_VC5.exe，CG_VC6.exe，CG_BC5.exe，CG_CB.exe，CG_CB3.exeを実行（自己解凍）する. 次節で示すディレクトリおよびファイルが作成されます.

(4) 2.2節でのEX2_1.cpp，EX2_2.cppの実行と同様にMK_PIC.cppを実行する.
本書のプログラム例で使用する画像が作成されます.

## 2．4　プログラム例の実行

プログラム例を実行するために，前節で説明したように環境ごとに必要なファイルを格納したディレクトリーを用意しています．Visual C++の場合はCG_VC4またはCG_VC5,CG_VC6，C++Builderの場合はCG_CBまたはCG_CB3，Borland C++の場合はCG_BC5です．

### 2．4．1　Visual C++の場合

Visual C++6.0の場合の実行手順は以下の通りです．

(1) ハードディスクのCG_VC6¥prj¥EX***n***.dswをダブルクリックする．
（Visual C++が図2-5のように起動する．***n***は章番号です）

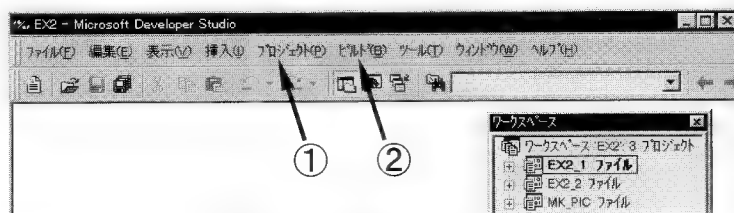



**図2-5**　Visual C++6.0の起動

(2) 図2-5の①でアクティブプロジェクトの設定を行う．（ EX2_1を選択する）
(3) 図2-5の②で"実行"を選択する．図2-6のようにウィンドウが開き，描画が行われます．
(4) PAUSEウィンドウの"OK"ボタンをクリックすると，ウィンドウが閉じてプログラムが終了します．ファイルメニューの"閉じる"を選択しても，プログラムは終了します．
　なお，図2-7に示すようにファイルメニューの"保存"を選択することによって，描画された画像をＢＭＰ形式でファイルに保存することができます．

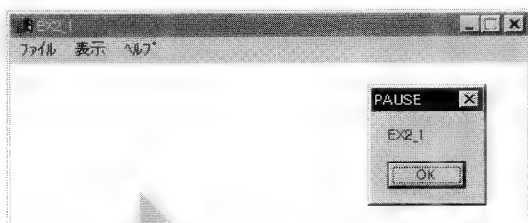
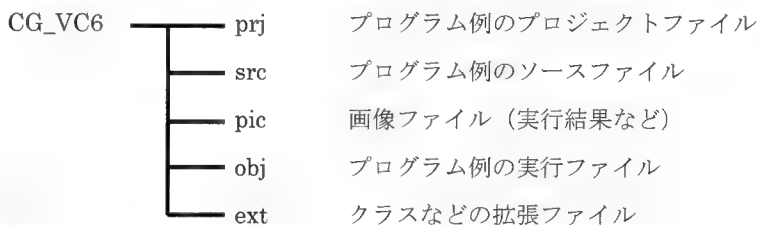



**図2-6**　プログラム例の実行　　**図2-7**　画像の保存

Visual C++5.0場合も，同様の操作で実行することができます．

Visual C++4.0場合は，操作が多少異なります．まず，CG_VC4¥prj¥EX***n***.mdpのダブルクリックから始め，プロジェクト(EX2_1 – Win32 Release)を選択して，実行(ビルド→実行)を行います．

ディレクトリ構成は以下の通りです．（Visual C++5.0および4.0も同様の構成）

## 2.4.2　C++ Builderの場合

C++ Builder3の場合の実行手順は以下の通りです．

(1) ハードディスクのCG_CB3¥prj¥EX***n***.bpgをダブルクリックする．
　（C++ Builderが図2-8のように起動する．***n***は章番号です）

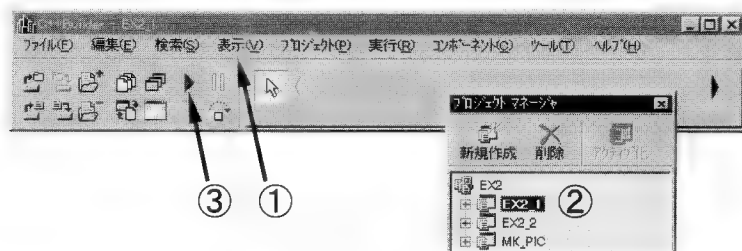



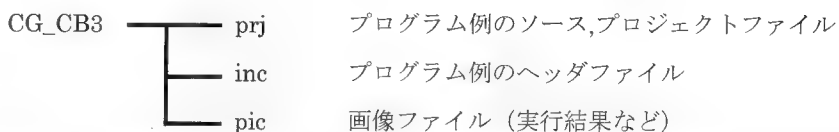
図2-8　C++ Builder3の起動

(2) 図2-8の表示メニュー①でプロジェクトマネージャを表示する．

(3) 図2-8のプロジェクトマネージャでプロジェクトを選択する．(EX2_1をダブルクリック)
　③のボタンをクリックして，プログラムを実行する．　図2-6のように描画が行われます．

(4) PAUSEウィンドウの操作，および画像の保存方法はVisual C++の場合と同様です．

C++ Builder場合は，操作が多少異なります．まず，CG_CB¥prj¥EX2_1.makのダブルクリックし，図2-8の③のボタンをクリックしてプログラムを実行します．

ディレクトリ構成は以下の通りです．（C++ Builderも同様の構成）

Borland C++5.0場合は，操作が異なります．まず，CG_BC5¥prj¥EX***n***.ideのダブルクリックから始め，プロジェクトウインドウを表示(表示→プロジェクト)して，実行(プロジェクトウィンドウ内のEX2_1.exeをダブルクリック)します．ディレクトリ構成は，Visual C++の場合と同様です．

リスト 2-1　Win クラスと Color クラス(Win.h)

```
 1: //    WINDOWS でグラフィックを行うためのライブラリ
 2: //                    98.02.10    by oga
 3:
 4: #ifndef   __WIN_H
 5: #define   __WIN_H
 6:
 7: #include  <stdio.h>
 8:
 9: #define   main()      g__main(int argc, char* argv[])
10: #define   Polygon     POlygon                          // Polygon は WINDOWS で定義済みのため
11: void  pause(char *txt="ＯＫで終了します！");           // 一時停止(ボタン入力待ち関数)
12: float max(float a, float b){ if(a>b)return a; return b;  }
13: float min(float a, float b){ if(a<b)return a; return b;  }
14:
15: //    ##############################################
16: //        Win,Color クラスに関係する記述
17: //    ##############################################
18:
19: //    カラーの定義
20: #define   WHITE   Color( 1.0, 1.0, 1.0 )  //  白
21: #define   YELLOW  Color( 1.0, 1.0, 0.0 )  //  黄色
22: #define   MAGENTA Color( 1.0, 0.0, 1.0 )  //  紫
23: #define   CYAN    Color( 0.0, 1.0, 1.0 )  //  水色
24: #define   RED     Color( 1.0, 0.0, 0.0 )  //  赤
25: #define   GREEN   Color( 0.0, 1.0, 0.0 )  //  緑
26: #define   BLUE    Color( 0.0, 0.0, 1.0 )  //  青
27: #define   BLACK   Color( 0.0, 0.0, 0.0 )  //  黒
28:
29: //    Color クラスの宣言
30: class Color{
31:       public:
32:       float   r,g,b;                                   //  ３原色の強度
33:       Color(float ir,float ig,float ib){  r=ir;   g=ig;   b=ib;   }
34:       Color()      {    r=g=b=0;     }
35: };
36:
37: //    Color クラス演算の宣言
38: Color operator*(float a,Color c){ return Color(a*c.r,a*c.g,a*c.b);     }
39: Color operator*(Color c,float a){ return Color(a*c.r,a*c.g,a*c.b);     }
40: Color operator/(Color c,float a){ return Color(c.r/a,c.g/a,c.b/a);     }
41: Color operator+(Color a,Color b){ return Color(a.r+b.r,a.g+b.g,a.b+b.b);  }
42:
43: //    Win クラスの宣言
44: class Win{
45:       public:
46:       int      work[28];                               //  作業領域
47:       int      wi,hi;                                  //  ウィンドウサイズ
48:       Win();                                           //  コンストラクタ
49:       ~Win()   {    close();    }                      //  デストラクタ
50:       void     open (char* wn,int x,int y,Color c=WHITE );//ウィンドウを開く
51:       void     color(Color c);                         //  描画色の設定
52:       void     point(int x,  int y);                   //  点を描画する
53:       Color    pixel(int x,  int y);                   //  描画色を返す
54:       void     move (int x,  int y);                   //  カーソルを移動する
55:       void     line (int x,  int y);                   //  線分を描画する
56:       void     line (int a[],  int n);                 //  複数線分を描画する
57:       void     paint(int a[],  int n);                 //  点の描画
58:       int      gprintf(char *format,...);
59:       void     read (char* fn);                        //  ファイルから画像を読込む
60:       void     clear();                                //  描画領域をクリアする
61:       void     close();                                //  ウィンドウを閉じる
62:  };
63: #endif
```

リスト 2-2　EX2_1.cpp

```
 1: //        簡単な図形を描く
 2: //        三角形と線分を描画する。
 3:
 4: #include        "win.h"
 5:
 6: main()
 7: {
 8:   Win w;                                       //  wというウィンドウを定義する
 9:   w.open ("EX2_1", 400,400);                   //  400×400のウィンドウを開く
10:
11:   w.color( Color(.9,.7,.3) );                  //  描画色を指定する
12:   int p[]={   100,100,     100,200,             //  三角形の座標を指定
13:               200,200,     100,100 };
14:   w.paint( p,4 );                              //  三角形を塗りつぶす
15:
16:   w.color( CYAN );                             //  描画色を指定する
17:   w.move  ( 200,300 );                         //  座標(200,300)
18:   w.line  ( 300,200 );                         //  座標(300,200)に線分を描画
19:
20:   pause("EX2_1");                              //  入力待ち
21: }
```

リスト 2-3　EX2_2.cpp

```
 1: //        簡単な図を画く(座標の演算を含む)
 2: //        リサジュー図形
 3:
 4: #include  "win.h"
 5: #include  "math.h"
 6:
 7: #define       PIR      (3.141952/180)
 8:
 9: main()
10: {
11:   Win w;
12:   w.open("EX2_2",400,400);                     //  ウィンドウ生成
13:   w.move ( 200, 200 );
14:
15:   int j=5;
16:   int k=2;
17:
18:   for(int i=0; i<360*k; i+=2) {
19:       float   r=100*sin(i*PIR*j/k)+30;         //  半径
20:       float   x=r*cos(i*PIR)+200;              //  X座標
21:       float   y=r*sin(i*PIR)+200;              //  Y座標
22:       float   t=fabs(r)/130;
23:       w.color( t*MAGENTA+(1-t)*YELLOW );
24:       w.line ( x, y );
25:   }
26:   pause("EX2_2");
27: }
```

# 3. 座標変換

この章では，2次元（平面）および3次元（空間）においての座標変換について考えていきます．その際，数学の教科書の記述に見られるような表現方法（例えば，点a，線分$l$，変換行列$t$など）を使用して，座標変換の計算（行列の計算）を行います．

## 3.1 2次元座標変換

スクリーン座標と標準座標（2次元）ついて考え，次に2次元（平面）で図形を移動あるいは変形を行う座標変換について考えていきます．

### 3.1.1 標準座標の使用

2章で述べたように，スクリーン座標（左上が座標(0,0)）を用いて描画を行う場合にWinクラスを使用しました．それでは，数学の教科書の記述のような標準座標を使用するにはどうしたらよいのでしょうか．本節ではこのことについて，考えていきます．

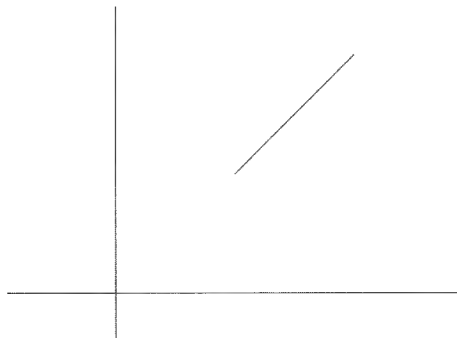



図3-1 線分と点の描画(EX3_1.cpp)

（1） スクリーン座標での描画

Winクラスを用いた場合のプログラム例(EX3_1.cpp)をリスト3-1に，実行結果を図3-1に示します．線分（標準座標(100,100)から標準座標(200,200)）と点（標準座標(100,200)）を描画するプログラム例です．図3-2に示すように，スクリーン座標はスクリーンの左上が(0,0)であり，標準座標とは異なります．そのため，標準座標を使用するためには，原点位置の設定や座標系の変換（標準座標からスクリーン座標への変換）の記述が必要となります．

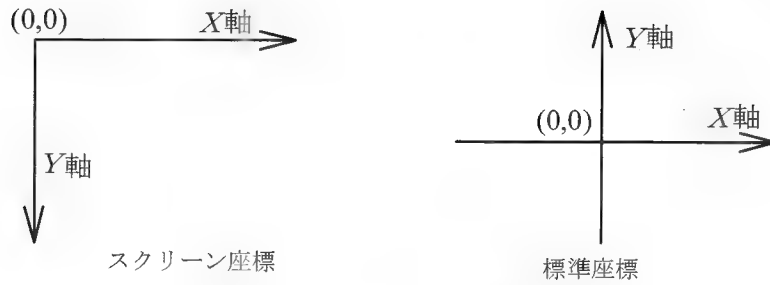



図3-2　スクリーン座標と標準座標

（2）　標準座標での描画

標準座標を用いて描画するには，描画する座標値をいったんスクリーン座標に変換してから描画します．式(3-1)を用いて，図3-3に示すように標準座標($x,y$)からスクリーン座標($x',y'$)に変換しています．標準座標の原点(0,0)は，スクリーン座標で($x_0,y_0$)の位置になります．

$$
\begin{aligned}
x' &= x + x_0 \\
y' &= -y + y_0
\end{aligned}
\tag{3-1}
$$

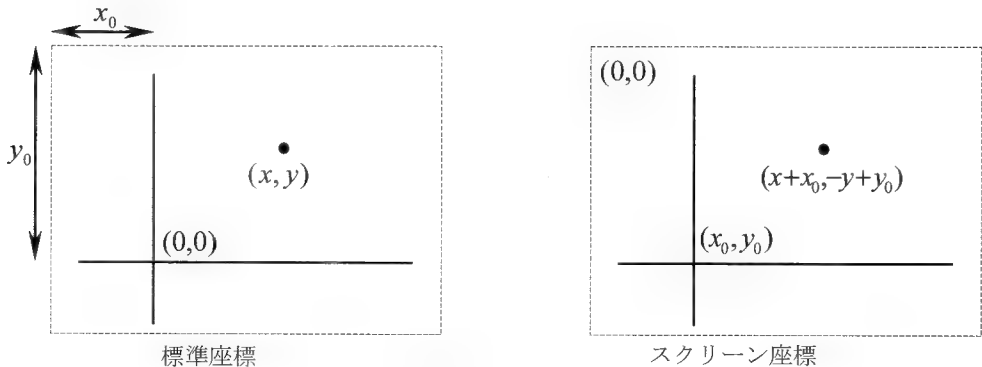



図3-3　標準座標からスクリーン座標へ

標準座標で描画するために，Win2Dクラス(内部でWinクラスを用いている)を用意しています．リスト3-1のプログラムを標準座標を用いて，書き直したプログラム例(EX3_2.cpp)をリスト3-2に示します．点および線分は，次項で示すPointクラス（点の記述），Lineクラス（複数線分の記述）のデータとして指定します．

Win2Dクラスを記述したヘッダファイルgraph1.hを巻末に示します．主なメンバ関数を以下に示します．標準座標の原点位置は，origin関数によってスクリーン座標で指定します．

```
Win2D(char* wn="Win2D", int x=WIDTH_DEF, int y=HEIGHT_DEF, Color c=WHITE);
                                   ウィンドウを開く
                                   (大きさ:WIDTH_DEF×HEIGHT_DEF, 背景:白)
void   origin(int x, int y);       原点位置をスクリーン座標(x,y)で設定する.
void   axis ();                    座標軸を描画する.
void   point(Point p);             点を描画する.
void   line (Point p0, Point p1);  線分を描画する.
void   line (Line& l);             線分を描画する.
```

なお，ヘッダファイルの内容は，本文で簡単に説明していますので，プログラミングにあまり立ち入らない場合には，解読する必要はありません．

### 3．1．2　点および線分の記述方法

数学の教科書の記述に見られるように，点pとか線分$l$のような表現を使用するにはどのようにしたらよいのでしょうか．それは，クラスを用いることによって可能となります．点の標準座標($x,y$)を表すPointクラスを考えます．Pointクラスはヘッダファイルpoint.h(巻末に示す)に記述されており，上述のgraph1.hにインクルードされて（含まれて）います（Pointクラスは3次元の座標が扱えるようにしていますが，Win2Dクラスは2次元なので$Z$座標は使用しません）．Pointクラスを用いることによって，座標(10,20)の点pを以下のように記述できるようになります．

```
Point p=Point(10,20);
Point p(10,20);
```

次に，Pointクラスを座標として用いて，複数の連結した線分を表すLineクラスを考えます．LineクラスはPointクラスの集合（配列）であり，データメンバ$n$がPoint数（始点，中継点，終点の数）を表しており，線分の数は$n-1$本となります．Lineクラスはヘッダファイルline.h(巻末に示す)に記述されており，上述のgraph1.hにインクルードされています．Lineクラスを用いることによって，座標(10,20)-(20,20)-(30,10)を結ぶ複数線分$l$を以下のように記述できるようになります．Point型の引数を最大10まで指定することができます．

```
Line  l=Line(Point(10,20), Point(20,20), Point(30,10));
Line  l(Point(10,20), Point(20,20), Point(30,10));
```

PointクラスおよびLineクラスではクラスの演算も記述しており，Pointクラス同士の演算(+,-)，PointクラスあるいはLineクラスと実数(float型)との演算(*,/)などを行うことができます．例えば，点aの座標値(10,20)と点bの座標値(30,30)の加算を点pとすると，以下のような記述ができるようになります（詳細は付録を参照して下さい）．

```
Point a=Point(10,20), b=Point(30,30);
Point p=a+b;
```

### 3．1．3　座標変換の方法

図形の幾何学的変換について，考えてみましょう．図形を別の位置に移動あるいは

別の図形に変形(拡大縮小を含む)することを，座標変換といいます．このとき，変換前（もと）の図形と変換後の図形との関係として，基本的なものを以下に示します．

① 平行移動（図3-4参照）
② 回転移動（図3-5参照）
③ 拡大縮小（図3-6参照）
④ 対称移動（図3-7参照）
⑤ 上記の合成

上記の変換は，一般に下記の式(3-2)によって行うことができます．座標$(x,y)$は図形を構成する点であり，座標$(x',y')$は変換後の図形を構成する点を示します．上記の変換は線形変換であるので，直線は直線に変換されます．すなわち，線分は線分に変換されることになり，線分の両端の点を式(3-2)で変換することによって，変換後の線分を得ることができます．変換の種類によって，$t_{0,0} \sim t_{1,2}$の各値は変わります．

$$x' = t_{0,0} \cdot x + t_{0,1} \cdot y + t_{0,2}$$
$$y' = t_{1,0} \cdot x + t_{1,1} \cdot y + t_{1,2} \qquad (3\text{-}2)$$

上記の式は，行列を用いて下式の通り記述することができます．このような変換を，数学的に２次元アフィン変換と呼んでいます．

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} t_{0,0} & t_{0,1} & t_{0,2} \\ t_{1,0} & t_{1,1} & t_{1,2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad (3\text{-}3)$$

$$p' = t * p \qquad (3\text{-}4)$$

$p = (x,y)$　　：変換前の座標（Pointクラス）

$p' = (x',y')$　　：変換後の座標（Pointクラス）

$$t = \begin{bmatrix} t_{0,0} & t_{0,1} & t_{0,2} \\ t_{1,0} & t_{1,1} & t_{1,2} \\ 0 & 0 & 1 \end{bmatrix}$$　：変換行列（TMatrixクラス）

結局，座標変換は式(3-4)で記述することができ，変換行列$t$を用いて行うことができます．プログラムでの変換行列の記述のために，TMatrixクラスを用意しています．TMatrixクラスはヘッダファイルtmatrix.h(巻末に示す)に記述されており，ヘッダファイルgraph1.hにインクルードされています（TMatrixクラスでは３次元の座標を扱えるように，変換行列を4行4列としています）．これによって，変換行列（TMatrixクラス）と点（Pointクラス）や線分（Lineクラス）の積，またTMatrixクラス同士の積を記述できるようにしています．例えば，変換行列$t$と点$a$や線分$l$の積は，下記

のように記述することができます．線分との積の場合は，線分を構成するすべての頂点座標を変換します．

```
TMatrix     t=TMatrix( 1.0,  0.0,  0.0,
                       0.0,  2.0,  0.0,
                       0.0,  0.0,  1.0);
Point       a=Point(10,10);
Point       p=t*a;
Line        l=Line(Point(10,10),Point(20,20),Point(10,20));
Line        m=t*l;
```

（1）　平行移動

平行移動（図3-4参照）は，図形の大きさおよび方向は変えずに移動することです．

移動後の座標$(x',y')$は，下式に示すようにそれぞれの移動値$(x_0,y_0)$を加算することによって求めることができます．

$$
\begin{aligned}
x' &= x + x_0 \\
y' &= y + y_0
\end{aligned}
\qquad (3\text{-}5)
$$

したがって，変換行列は下記のようになります．

$$
t = \begin{bmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{bmatrix}
\qquad (3\text{-}6)
$$

（2）　回転移動

回転移動（図3-5参照）は，原点を中心に回転することであり，$\theta$度回転する場合の移動後の座標$(x',y')$は，下式によって求めることができます．

$$
\begin{aligned}
x' &= x \cdot (-\sin\theta) + y \cdot \cos\theta \\
y' &= x \cdot \cos\theta + y \cdot \sin\theta
\end{aligned}
\qquad (3\text{-}7)
$$

したがって，変換行列は下記のようになります．

$$
t = \begin{bmatrix} -\sin\theta & \cos\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}
\qquad (3\text{-}8)
$$

平行移動と回転移動のプログラム例(EX3_3.cpp)をリスト3-3に，実行結果を図3-4および図3-5に示します．プログラムを実行するとウィンドウが2つ開き，平行移動の結果は，最初に開くウィンドウw1に以下の通り描画されます．

　変換前の図形：水色（Y字形の図形）

変換後の図形：赤色（$X$方向に－100の平行移動）

変換後の図形：紫色（$X$方向に－50，$Y$方向に－50の平行移動）

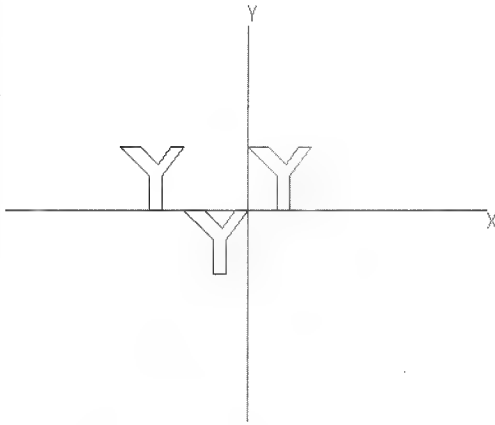
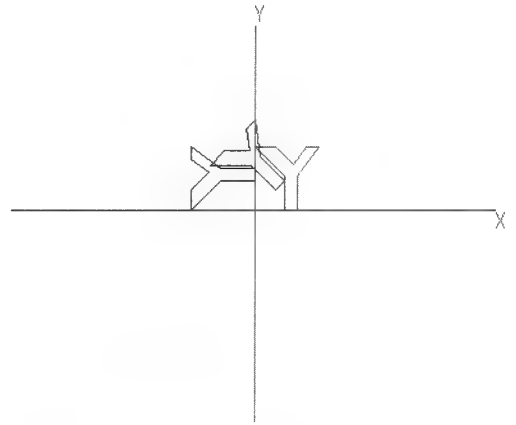



**図3-4**　平行移動(EX3_3.cpp:w1)　　**図3-5**　回転移動(EX3_3.cpp:w2)

Y字形図形の各頂点の変換を行い，変換後の各頂点を線分で結ぶことによって，変換後の図形は得られます．このプログラムでは，"w1.line(t*l);"（$t$：変換行列，$l$はY字型の線分）の記述で上記の変換後の図形（Y字型を$t$で変換した図形）が描画されます．

回転移動の結果は，ウィンドウw2に以下の通り描画されます．

変換前の図形：水色（Y字形の図形）

変換後の図形：赤色（45度回転）

変換後の図形：紫色（90度回転）

（3）　スケーリング

スケーリング（図3-6参照）は，原点を中心に拡大（縮小）することであり，$X$成分を$a$倍$Y$成分を$b$倍する場合の変換後の座標$(x',y')$は，下式によって求めることができます．

$$
\begin{aligned}
x' &= a \cdot x \\
y' &= b \cdot y
\end{aligned}
\quad (3\text{-}9)
$$

したがって，変換行列は下記のようになります．

$$
t = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (3\text{-}10)
$$

（4）　対称変換

$X$軸対象，$Y$軸対象，原点対象とする場合（図3-7参照），変換後の座標$(x',y')$はスケーリングの計算と同様に，下式によって求めることができます．

$$x' = a \cdot x$$
$$y' = b \cdot y \tag{3-11}$$

したがって，変換行列は下記のようになります．

$$t = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3-12}$$

パラメータ$a$，$b$と変換の関係を以下に示します．

① $X$軸対象：$a=-1$，$b=1$
② $Y$軸対象：$a=1$，$b=-1$
③ 原点対象：$a=-1$，$b=-1$

スケーリングと対象変換のプログラム例(EX3_4.cpp)をリスト3-4に，実行結果を図3-6および図3-7に示します．プログラムを実行するとウィンドウが2つ開き，スケーリングの結果は，ウィンドウw1に以下の通り描画されます．

変換前の図形：水色（Y字形の図形）
変換後の図形：赤色（$X$方向2倍，$Y$方向2倍）
変換後の図形：紫色（$X$方向1倍，$Y$方向3倍）

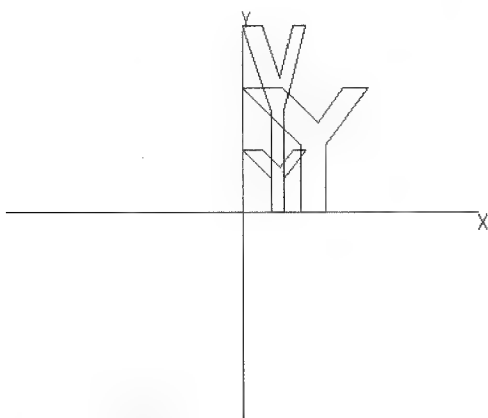
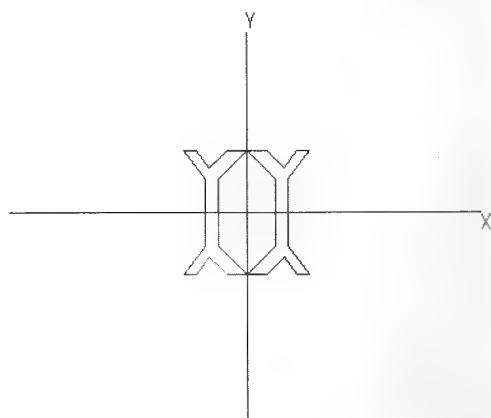



**図3-6**　スケーリング(EX3_4.cpp:w1)　　**図3-7**　対称変換(EX3_4.cpp:w2)

対象変換の結果は，ウィンドウw2に以下の通り描画されます．

変換前の図形：水色（Y字形の図形）

変換後の図形：赤色（$X$軸対象）

変換後の図形：紫色（$Y$軸対象）

変換後の図形：黄色（原点対象）

なお，平行移動，回転移動，スケーリングの変換行列を生成する関数として，tmatrix.hに以下の関数を用意しています．

```
TMatrix      move(Point& a);                    平行移動
TMatrix      rotate(float a);                   回転移動
TMatrix      scale(float a, float b);           スケール変換
```

### 3.1.4　座標変換の合成

複数の変換を行う場合について，考えてみましょう．例として，回転変換（$t_1$）の後に平行移動（$t_2$）を行う場合，下式で変換（$t_1$の変換の後に$t_2$の変換）を行うことができます．

$$p' = t_2 * t_1 * p = t * p$$
$$t = t_2 * t_1 \qquad (3\text{-}13)$$

$p$　：変換前の座標

$p'$　：変換後の座標

$t_1$　：回転変換行列

$t_2$　：平行移動変換行列

すなわち，合成した変換行列$t$を計算（$t_1$と$t_2$の積）し，その変換行列で座標変換を行うことによって，２つの変換（回転変換の後に平行移動）を行うことができます．変換が３つ以上の場合でも，その変換行列の積をとることによって，同様に計算することができます．

上記の方法（合成した変換行列を使用する）により，任意の位置での回転変換の例を次に示します．

（１）　任意の位置での回転変換

任意の位置$c$を中心にした回転変換は，以下の３段階の操作（①～③）で行うことができます．

① 平行移動（$-c$）を行い，回転の中心位置を原点に移動する．

② 原点を中心に回転変換を行う．

③ 平行移動（$c$）を行い，回転の中心位置をもとの位置に戻す．

すなわち，任意の位置での変換は，①回転位置を原点に平行移動して，②原点で目的の変換を行い，③平行移動でもとの位置に戻すことにより行うことができます．プログラム例（EX3_5.cpp）をリスト3-5に，実行結果を図3-8に示します．プログラム中では，上記①，②，③の各変換を合成した変換行列$t$を求めて使用しています．

変換前の図形：水色（Y字形の図形）

変換後の図形：赤色（30度回転）
変換後の図形：紫色（60度回転）
変換後の図形：黄色（90度回転）

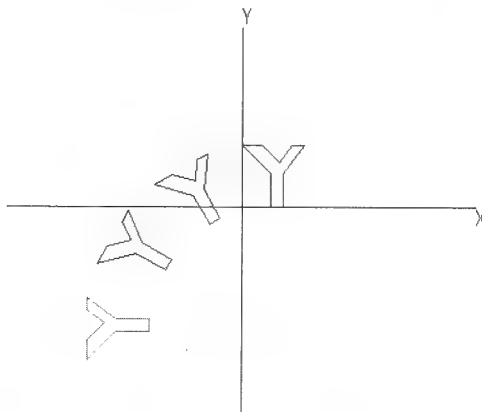



図3-8　回転移動(EX3_5.cpp)

## 3．2　3次元座標変換

前項で述べた2次元座標変換を3次元座標へ拡張します．まず，点，線分の記述方法および変換の方法について説明します．次に，3次元座標変換について説明します．基本的な考え方は同じですが，3次元になるので$Z$座標の成分が増えることになります．なお，3次元空間の図形（点，線）をスクリーンに描画するには，3次元視野変換を行う必要があります．3次元視野変換については，次章で考えることにします．

### 3．2．1　点および線分の記述方法

前項では，描画を行うためにWin2Dクラスを使用していました．そこで，Win3Dクラスを用意して，3次元座標をスクリーン座標に変換して，Winクラスを用いて描画を行います．Win3Dクラスは，前項で使用していたWin2Dクラスを3次元用に拡張したものであり，基本的な考え方は同じです．また，関数も類似していますが，引数が3次元になっています．主に使用する関数を以下に示します．Win3Dが記述されているgraph2.hを巻末に示します．Win3Dクラスの仕組みについては，次章で説明します．

```
Win3D(char* wn="Win3D",int x=WIDTH_DEF, int y=HEIGHT_DEF, Color c=WHITE);
                                    ウィンドウを開く
                                    (大きさ:WIDTH_DEF×HEIGHT_DEF,背景:白)
void  origin(int x, int y);         原点位置をスクリーン座標(x,y)に設定する.
void  axis ();                      座標軸を描画する.
Void  setview(float a, float e, float d=1000); 視点を指定（方向および距離)
void  point(Point p);               点を描画する.
void  line (Point p0, Point p1);    線分を描画する.
```

```
void  line (Line& l);                    線分を描画する.
```

（1）　Pointクラス

前項で述べたPointクラスと同じものです．なお，3次元ベクトルを記述するために"Vector"をdefine文で"Point"に置換しており，Vectorクラスがあるように見せかけています．したがって，プログラム中のVectorとPointは同じものです．2次元の場合と同様に，下記のように点Pを記述することができます．また，Pointクラスでは，ベクトルの演算ができるように加減算（+，−），内積（*），外積（%）の演算も定義しています（内積と外積の計算については付録を参照）．

```
Point  p=Point(10,20,30);
Point  q=Point(20,20,10);
Point  r=p%q;                              外積の計算
```

（2）　Lineクラス

前項で述べたLineクラスと同じもので，複数の連結した線分を表します．2次元の場合と同様に，下記のように線分$l$を記述することができます．

```
Line   l=Line(Point(10,20,0), Point(20,20,0), Point(30,10,10));
Line   l(Point(10,20,0), Point(20,20,0), Point(30,10,10));
```

ヘッダファイルgraph2.hには，PointクラスおよびLineクラスを記述しているヘッダファイルpoint.hおよびline.hがインクルードされており，前述のように3次元空間に点や線分を描画できるようにしています．視点の方向は方位角30度，仰角30度（省略値）としています（setview関数で指定できます）．

### 3.2.2　座標変換の方法

2次元の場合を，3次元に拡張して考えることができます．ある図形が変換により，別の図形あるいは別の位置に移動されます．このとき，変換前（もと）の図形と変換後の図形との関係として，基本的なものを以下に示します．

① 平行移動
② 回転移動
③ 拡大縮小
④ 対称移動
⑤ 上記の合成

これらの変換は2次元の場合と同様に，一般に下記の式によって行うことができます．座標$(x,y,z)$は図形を構成する点であり，座標$(x',y',z')$は変換後の図形を構成する点を示します．変換の種類によって，$t_{0,0}$～$t_{2,3}$の各値は決まります．

$$
\begin{aligned}
x' &= t_{0,0}\cdot x + t_{0,1}\cdot y + t_{0,2}\cdot z + t_{0,3} \\
y' &= t_{1,0}\cdot x + t_{1,1}\cdot y + t_{1,2}\cdot z + t_{1,3} \\
z' &= t_{2,0}\cdot x + t_{2,1}\cdot y + t_{2,2}\cdot z + t_{2,3}
\end{aligned}
\tag{3-14}
$$

上記の式は，行列を用いて下式のように記述することができます．このような変換を，数学的に3次元アフィン変換と呼んでいます．

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} t_{0,0} & t_{0,1} & t_{0,2} & t_{0,3} \\ t_{1,0} & t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,0} & t_{2,1} & t_{2,2} & t_{2,3} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{3-15}$$

$$p' = t * p \tag{3-16}$$

$p = (x,y,z)$　　：変換前の座標（Pointクラス）

$p' = (x',y',z')$　　：変換後の座標（Pointクラス）

$$t = \begin{bmatrix} t_{0,0} & t_{0,1} & t_{0,2} & t_{0,3} \\ t_{1,0} & t_{1,1} & t_{1,2} & t_{1,3} \\ t_{2,0} & t_{2,1} & t_{2,2} & t_{2,3} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$　：変換行列（TMatrixクラス）

結局，座標変換は式(3-16)で記述することができ，変換行列$t$を用いて行うことができます．

プログラムでの変換行列の記述のために，TMatrixクラスを用意しています．TMatrixクラスはtmatrix.h(巻末に示す)としてヘッダファイルgraph2.hにインクルードされています．2次元の場合と同様に，点aや線分$l$の座標変換は，下記のように記述することができます．

```
TMatrix t=Tmatrix(1.0,  0.0,  0.0,  0.0
                  0.0,  2.0,  0.0,  0.0
                  0.0,  0.0,  1.0,  0.0
                  0.0,  0.0,  0.0,  1.0);
Point   a=Point(10,10,10);
Point   p=t*a;
Line    l=Line(Point(10,10,10).Point(20,20,10),Point(10,20,10));
Line    m=t*l;
```

（1）　平行移動

平行移動では，移動後の座標$p' = (x',y',z')$は，移動前の座標$p = (x,y,z)$に移動値$(x_0,y_0,z_0)$を加算することによって求められます．

$$\begin{aligned} x' &= x + x_0 \\ y' &= y + y_0 \\ z' &= z + z_0 \end{aligned} \tag{3-17}$$

したがって，変換行列は下記のようになります．

$$t=\begin{bmatrix}1 & 0 & 0 & x_0\\ 0 & 1 & 0 & y_0\\ 0 & 0 & 1 & z_0\\ 0 & 0 & 0 & 1\end{bmatrix} \tag{3-18}$$

（2）　回転移動

3次元では，回転軸が$X$軸，$Y$軸，$Z$軸の場合がに考えられます．各場合の回転を図3-9に示す．

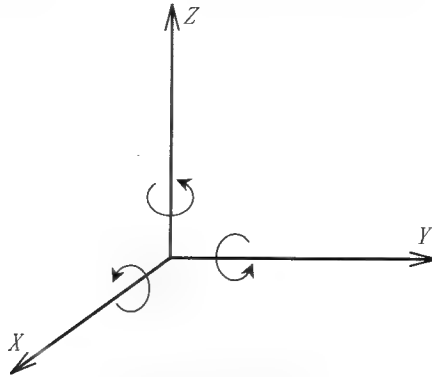



**図3-9**　回転の方向

①　$X$軸を中心に$\theta$度回転する場合

2次元の場合は$XY$平面での原点を中心とする回転でしたが，$X$軸を中心に回転する場合は2次元での$YZ$平面での回転に相当します．すなわち，2次元の場合の$X$を$Y$，$Y$を$Z$と読み替えることによって$Y$座標と$Z$座標の変換式が求められます．$X$座標は変化しません．そのため，移動後の座標$p'=(x',y',z')$は，下式によって求められます．

$$\begin{aligned} x' &= x\\ y' &= y\cdot\cos\theta - z\cdot\sin\theta\\ z' &= y\cdot\sin\theta + z\cdot\cos\theta \end{aligned} \tag{3-19}$$

したがって，変換行列は下記のようになります．

$$t=\begin{bmatrix}1 & 0 & 0 & 0\\ 0 & \cos\theta & -\sin\theta & 0\\ 0 & \sin\theta & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \tag{3-20}$$

②　$Y$軸を中心に$\theta$度回転する場合

①の場合と同様に考えることができ，変換行列は下記のようになります．

$$t=\begin{bmatrix}\cos\theta & 0 & \sin\theta & 0\\ 0 & 1 & 0 & 0\\ -\sin\theta & 0 & \cos\theta & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \tag{3-21}$$

③　$Z$軸を中心に$\theta$度回転する場合

①の場合と同様に考えることができ，変換行列は下記のようになります．

$$t=\begin{bmatrix}\cos\theta & -\sin\theta & 0 & 0\\ \sin\theta & \cos\theta & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix} \tag{3-22}$$

平行移動と回転移動を行うプログラム例（EX3_6.cpp）をリスト3-6に，実行結果を図3-10および図3-11に示します．このプログラムを実行するとウィンドウが２つ開き，平行移動の結果はウィンドウw1に以下の通り描画されます．

変換前の図形：水色（Y字型の図形）

変換後の図形：赤色（$Z$方向に100の平行移動）

変換後の図形：紫色（$X$方向に－150の平行移動）

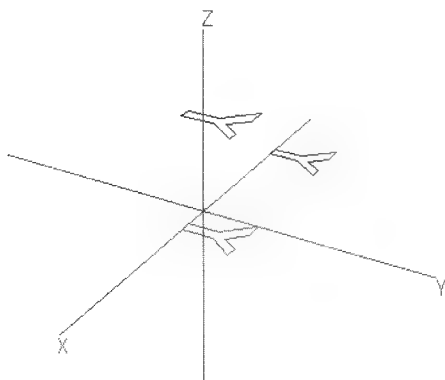
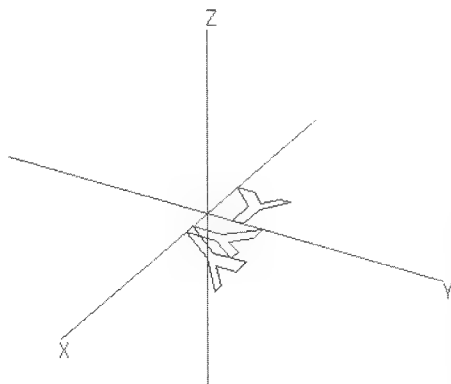



**図3-10**　平行移動（EX3_6.cpp:w1）　　**図3-11**　回転移動（EX3_6.cpp:w2）

回転移動の結果は，ウィンドウw2に以下の通り描画されます．

変換前の図形：水色（Y字型の図形）

変換後の図形：赤色（$Z$軸を中心に90度の回転移動）

変換後の図形：紫色（$X$軸を中心に－45度の回転移動）

（３）　スケーリング

スケーリングは，原点を中心に拡大（縮小）することであり，X成分を$a$倍，Y成分を$b$倍，Z成分を$c$倍する場合の移動後の座標$p'=(x',y',z')$は，下式によって求めることができます．

$$
\begin{aligned}
x' &= a \cdot x \\
y' &= b \cdot y \\
z' &= c \cdot z
\end{aligned}
\tag{3-23}
$$

したがって，変換行列は下記のようになります．

$$
t = \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\tag{3-24}
$$

4） 対称変換

対称変換として，点，直線，平面に対するものがあります．スケーリングと同様の変換式で変換を行うことができます．パラメータ$a,b,c$の各値を1あるいは-1とすることによって，どのような変換になるかを示します．

パラメータ$a,b,c$の値と対称変換の種類

| | $a=1$ | | $a=-1$ | |
|---|---|---|---|---|
| | $b=1$ | $b=-1$ | $b=1$ | $b=-1$ |
| $c=1$ | 無変換 | XZ平面 | YZ平面 | Z軸 |
| $c=-1$ | XY平面 | X軸 | Y軸 | 原点 |

スケーリングおよび対称変換を行うプログラム例（EX3_7.cpp）をリスト3-7に，実行結果を図3-12および図3-13に示します．このプログラムを実行するとウィンドウが2つ開き，スケーリングの結果はウィンドウw1に以下の通り描画されますもとの図形（XY平面上のY字形）を拡大した図形を描画します．

変換前の図形：水色（Y字型の図形）

変換後の図形：赤色（各方向に3倍のスケーリング）

変換後の図形：紫色（X方向に0.5倍，Y方向に2倍）

対称変換の結果は，ウィンドウw2に以下の通り描画されます．元の図形（XY平面上のY字形）をYZ平面対称およびZ軸対称の図形を描画しています．

変換前の図形：水色（Y字型の図形）

変換後の図形：赤色（YZ平面対称）

変換後の図形：紫色（$Z$軸対称）

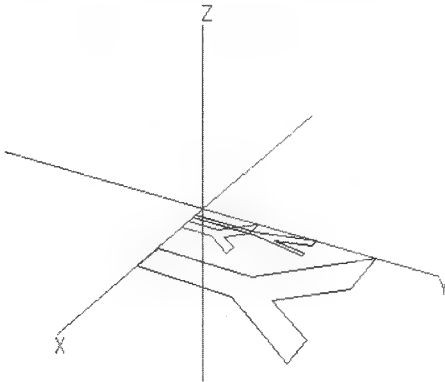
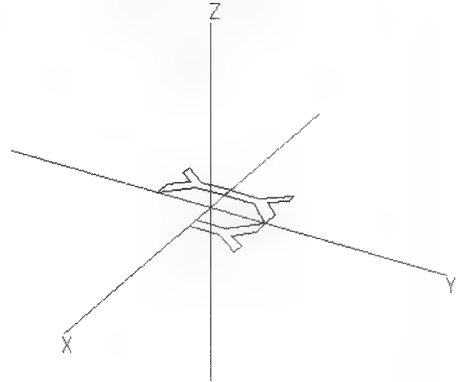



**図3-12**　スケーリング(EX3_7.cpp:w1)　　**図3-13**　対称変換(EX3_7.cpp:w2)

なお，平行移動，回転移動，スケーリング，対称変換の変換行列を生成する関数として，tmatrix.hに以下の関数を用意しています．

```
TMatrix   move(Point a) ;                   平行移動
TMatrix   rot_x(float r);                   X軸を中心とする回転移動(r度)
TMatrix   rot_y(float r);                   Y軸を中心とする回転移動(r度)
TMatrix   rot_z(float r);                   Z軸を中心とする回転移動(r度)
TMatrix   rotate(Point& a, float r);        回転軸を指定した回転移動(r度)
TMatrix       scale(float a, float b, float c);     スケーリング
```

### ３．２．３　座標変換の合成

３次元の座標変換の合成は，２次元の場合と同様に考えることができ，変換行列の積として計算することができます．$n$回の変換を行う場合には，下式で計算をすることができます．

$$p' = t * p$$
$$t = t_n * \cdots * t_2 * t_1 \quad (3\text{-}25)$$

$p$　：変換前の座標

$p'$　：変換後の座標

$t_i$　：$i$番目の変換行列（$i = 1,2,\cdots,n$）

（１）　原点を通る直線を中心とする回転変換

回転軸の方向ベクトルを$a$，回転角を$b$とすると，以下のような手順（①～⑤）で変換を行うことができます．

① 方向ベクトル$a$を，$XZ$平面上へ$Z$軸回転して移動する（変換行列は$t_1$）．

② ①で移動した方向ベクトルを$Y$軸回転して，$X$軸上に移動する（変換行列は$t_2$）．

③ X軸回転（回転角$b$）を行う（変換行列は$t_3$）．

④ ②の逆変換を行う（変換行列は$t_4$）．

⑤ ①の逆変換を行う（変換行列は$t_5$）．

すなわち，①,②方向ベクトル$a$を$X$軸へ回転移動し，③目的とする回転を行い，④，⑤もとの方向に回転移動する（①,②の逆）ことによって，任意の直線（方向ベクトル$a$）を中心とする回転変換を行うことができます．

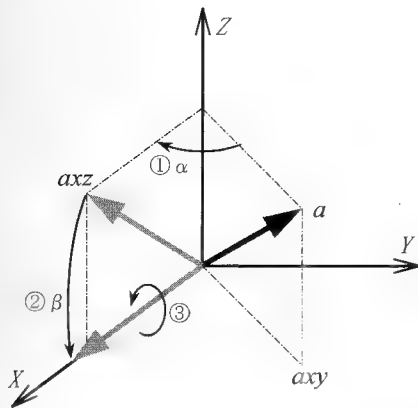



**図3-14**　原点を通る直線を中心とする回転変換

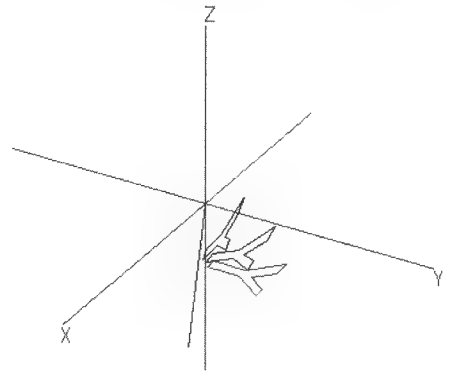



**図3-15**　回転移動(EX3_8.cpp)

図3-14に変換の手順を示します．図中の$axy$，$axz$の座標によって，①および②の回転角を知ることができます．①の回転角を$\alpha$，②の回転角を$\beta$とすれば，以下の関係が成り立ちます．

$$\sin\alpha = -axy.y/|axy| \qquad \sin\beta = axz.z$$
$$\cos\alpha = axy.x/|axy| \qquad \cos\beta = axz.x$$

プログラム例(EX3_8.cpp)をリスト3-8に，実行結果を図3-15に示します．回転軸のベクトル$a$は(2,1,0)とします．

変換前の図形：水色（Y字型の図形）

変換後の図形：赤色（30,60度回転）

回転軸　　　：青色

プログラム中では，単位ベクトルを返すunit関数，および正弦と余弦の引数で回転角を指定するrot_y,rot_z関数を使用しています．

```
Point      unit(Point& a);                単位ベクトルを返す
TMatrix    rot_y(float sn,float cs);      Y軸回転(sn:sin(r),cs:cos(r))
TMatrix    rot_z(float sn,float cs);      Z軸回転(sn:sin(r),cs:cos(r))
```

## 演習問題

**3-1**　任意点における回転移動

Ｙ字型図形を，点(0，－100，0)を中心に$X$軸回転(30，60，90度)せよ．

**3-2**　らせん状の回転移動

Ｙ字型図形を，$Y$軸方向に50移動する．その図形を$Z$軸回転($360t/N$度)し，$Z$軸方向に$100t/N$移動せよ（$N=36$，$t=-N \sim N$，複数の図形を描画する）．

**3-3**　らせん状の回転移動

$YZ$平面上の長方形（60×40）を，以下のように変換せよ．

①　$X$軸回転する($360t/N$度)．

②　$Y$軸方向に130移動する．

③　$Z$軸回転する($360t/N$度)．

$t=0 \sim N-1$，$N=150$とする．

リスト 3-1　EX3_1.cpp

```
//        簡単な図形を描く
//        標準座標(100,100)から(200,200)までの線分と点(200,100)を描画する。

#include        "win.h"

main()
{
    Win w;
    w.open("EX3_1", 400,300);                          //  ウィンドウを開く

    int origin_x=100;                                  //  原点の指定
    int origin_y=250;

    w.color( CYAN );                                   //  座標軸の描画
    w.move  ( 10,origin_y );                           //  X軸の描画
    w.line  ( 399-10,origin_y );
    w.move  ( origin_x,10 );                           //  Y軸の描画
    w.line  ( origin_x,299-10 );

    w.color( RED );
    //        標準座標からスクリーン座標への変換
    w.move  ( origin_x+100,origin_y-100 );             //  座標(100,100)
    w.line  ( origin_x+200,origin_y-200 );             //  座標(200,200)に線分を描画
    w.point( origin_x+200,origin_y-100 );              //  座標(200,100)に点を描画

    pause("EX3_1");
}
```

リスト 3-2　EX3_2.cpp

```
 1: //        簡単な図を画く
 2: //        標準座標(100,100)から(200,200)までの線分と点(200,100)を描画する。
 3:
 4: #include        "graph1.h"
 5:
 6: main()
 7: {
 8:     Win2D   w("EX3_2",400,300);                    //  ウィンドウ生成
 9:     w.origin(100,250);                             //  原点位置の設定
10:     w.axis();                                      //  座標軸の表示
11:
12:     Line    l=Line( Point(100,100),Point(200,200) );
13:     w.color( RED );                                //  描画色の設定
14:     w.line  ( l );                                 //  座標(100,100)から(200,200)に線分
15:     w.point( Point(200,100) );                     //  座標(200,100)に点を描画
16:
17:     pause("EX3_2");
18: }
```

リスト 3-3　EX3_3.cpp

```
 1: //        簡単な図形の変換（座標の演算を含む）
 2:
 3: #include        "graph1.h"
 4:
 5: main()
 6: {
 7:     TMatrix t;                                     //  変換行列
 8:     Line    l=Line( Point(0,50),    Point(23,27),  //  Y字形の頂点座標の設定
 9:                     Point(23,0),    Point(33,0),
10:                     Point(33,27),   Point(50,50),
11:                     Point(40,50),   Point(30,36),
```

```
12:                       Point(16,50),    Point(0,50) );
13:
14: //   Ｙ字型図形の平行移動
15:     Win2D   w1("w1:平行移動");                          //  ウィンドウ生成
16:     w1.axis ();                                         //  座標軸の表示
17:     w1.color( CYAN );                                   //  描画色の設定
18:     w1.line ( l );                                      //  Ｙ字形の描画
19:
20:     t=TMatrix(  1,       0,       -100,                 //  変換行列の設定（平行移動１）
21:                 0,       1,       0,                    //  Ｘ方向に-100
22:                 0,       0,       1   );
23:     w1.color( RED );                                    //  描画色の設定
24:     w1.line ( t*l );                                    //  変換して描画
25:
26:     t=TMatrix(  1,       0,       -50,                  //  変換行列の設定（平行移動２）
27:                 0,       1,       -50,                  //  Ｘ方向に-50、Ｙ方向に-50
28:                 0,       0,       1   );
29:     w1.color( MAGENTA );                                //  描画色の設定
30:     w1.line ( t*l );                                    //  変換して描画
31:
32:
33: //   Ｙ字型図形の回転変換
34:     Win2D   w2("w2:回転変換");                          //  ウィンドウ生成
35:     w2.axis ();                                         //  座標軸の表示
36:     w2.color( CYAN );                                   //  描画色の設定
37:     w2.line ( l );                                      //  Ｙ字形の描画
38:
39:
40:     float   a=PIR*45;
41:     t=TMatrix(  cos(a),-sin(a),      0,                 //  変換行列の設定（回転移動１）
42:                 sin(a), cos(a),      0,                 //  45 度回転
43:                 0,             0,       1   );
44:     w2.color( RED );                                    //  描画色の設定
45:     w2.line ( t*l );                                    //  変換して描画
46:
47:     a=PIR*90;
48:     t=TMatrix(  cos(a),-sin(a), 0,                      //  変換行列の設定（回転移動２）
49:                 sin(a), cos(a), 0,                      //  90 度回転
50:                 0,      0,       1   );
51:     w2.color( MAGENTA );                                //  描画色の設定
52:     w2.line ( t*l );                                    //  変換して描画
53:
54:     pause("EX3_3");
55: }
```

リスト 3-4　EX3_4.cpp

```
 1: //        簡単な図形の変換（座標の演算を含む）
 2:
 3: #include         "graph1.h"
 4:
 5: main()
 6: {
 7:     TMatrix t;                                          //  変換行列
 8:     Line    l=Line( Point(0,50),     Point(23,27),      //  Ｙ字形の頂点座標の設定
 9:                     Point(23,0),     Point(33,0),
10:                     Point(33,27),    Point(50,50),
11:                     Point(40,50),    Point(30,36),
12:                     Point(16,50),    Point(0,50) );
13:
14: //   Ｙ字型図形のスケーリング
15:     Win2D   w1("w1:スケーリング");                      //  ウィンドウ生成
16:     w1.axis ();                                         //  座標軸の表示
17:     w1.color( CYAN );                                   //  描画色の設定
18:     w1.line ( l );                                      //  Ｙ字形の描画
19:
```

```
20:     t=TMatrix(  2,      0,      0,                        //  変換行列の設定
21:                 0,      2,      0,                        //  X を 2 倍、Y を 2 倍
22:                 0,      0,      1   );
23:     w1.color( RED );                                      //  描画色の設定
24:     w1.line ( t*l );                                      //  変換して描画
25:
26:     t=TMatrix(  1,      0,      0,                        //  変換行列の設定
27:                 0,      3,      0,                        //  X を 1 倍、Y を 3 倍
28:                 0,      0,      1   );
29:     w1.color( MAGENTA );                                  //  描画色の設定
30:     w1.line ( t*l );                                      //  変換して描画
31:
32:
33: //  Y字型図形の対象変換
34:     Win2D   w2("w2:対象変換");                            //  ウィンドウ生成
35:     w2.axis ();                                           //  座標軸の表示
36:     w2.color( CYAN );                                     //  描画色の設定
37:     w2.line ( l );                                        //  Y字形の描画
38:
39:     t=TMatrix(  1,      0,      0,                        //  変換行列の設定
40:                 0,      -1,     0,                        //  Y 軸対象
41:                 0,      0,      1   );
42:     w2.color( RED );                                      //  描画色の設定
43:     w2.line ( t*l );                                      //  変換して描画
44:
45:     t=TMatrix(  -1,     0,      0,                        //  変換行列の設定
46:                 0,      1,      0,                        //  X 軸対象
47:                 0,      0,      1   );
48:     w2.color( MAGENTA );                                  //  描画色の設定
49:     w2.line ( t*l );                                      //  変換して描画
50:
51:     t=TMatrix(  -1,     0,      0,                        //  変換行列の設定
52:                 0,      -1,     0,                        //  原点対象
53:                 0,      0,      1   );
54:     w2.color( YELLOW );                                   //  描画色の設定
55:     w2.line ( t*l );                                      //  変換して描画
56:
57:     pause("EX3_4");
58: }
```

リスト 3-5　EX3_5.cpp

```
 1: //      簡単な図形の変換（任意の位置での回転）
 2:
 3: #include    "graph1.h"
 4:
 5: main()
 6: {
 7:     TMatrix t;                                            //  変換行列
 8:     Line    l=Line( Point(0,50),    Point(23,27),         //  Y字形の頂点座標の設定
 9:                     Point(23,0),    Point(33,0),
10:                     Point(33,27),   Point(50,50),
11:                     Point(40,50),   Point(30,36),
12:                     Point(16,50),   Point(0,50) );
13:
14:     //  Y字型図形の任意の位置での回転
15:     Win2D   w1("EX3_5");                                  //  ウィンドウ生成
16:     w1.axis ();                                           //  座標軸の表示
17:     w1.color( CYAN );                                     //  描画色の設定
18:     w1.line ( l );                                        //  Y字形の描画
19:
20:     Point   c=Point( 25, -100);                           //  回転中心位置
21:     w1.color(BLUE);
22:     w1.point(c);
23:
24:     t=move(c)*rotate(30)*move(-1*c);                      //  変換行列の計算
```

```
25:     w1.color( RED );                                     //  描画色の設定
26:     w1.line ( t*l );                                     //  変換して描画
27:
28:     t=move(c)*rotate(60)*move(-1*c);                     //  変換行列の計算
29:     w1.color( MAGENTA );                                 //  描画色の設定
30:     w1.line ( t*l );                                     //  変換して描画
31:
32:     t=move(c)*rotate(90)*move(-1*c);                     //  変換行列の計算
33:     w1.color( YELLOW );                                  //  描画色の設定
34:     w1.line ( t*l );                                     //  変換して描画
35:
36:     pause("EX3_5");
37: }
```

リスト3-6　EX3_6.cpp

```
 1: //       ３次元空間の図形の変換（座標の演算を含む）
 2:
 3: #include        "graph2.h"
 4:
 5: main()
 6: {
 7:     TMatrix t;                                           //  変換行列
 8:     Line    l=Line( Point(0,50,0),      Point(23,27,0),  //  Ｙ字形の頂点座標の設定
 9:                     Point(23,0,0),      Point(33,0,0),
10:                     Point(33,27,0),     Point(50,50,0),
11:                     Point(40,50,0),     Point(30,36,0),
12:                     Point(16,50,0),     Point(0,50,0) );
13:
14: //  Ｙ字型図形の平行移動
15:     Win3D   w1("w1:平行移動");                           //  ウィンドウ生成
16:     w1.axis ();                                          //  座標軸の表示
17:     w1.color( CYAN );                                    //  描画色の設定
18:     w1.line ( l );                                       //  Ｙ字形の描画
19:
20:     t=TMatrix(  1,      0,      0,      0,               //  変換行列の設定
21:                 0,      1,      0,      0,               //  Z 方向に 100
22:                 0,      0,      1,      100,
23:                 0,      0,      0,      1   );
24:     w1.color( RED );                                     //  描画色の設定
25:     w1.line ( t*l );                                     //  変換して描画
26:
27:     t=TMatrix(  1,      0,      0,      -150,            //  変換行列の設定
28:                 0,      1,      0,      0,               //  X 方向に-150
29:                 0,      0,      1,      0,
30:                 0,      0,      0,      1   );
31:     w1.color( MAGENTA );                                 //  描画色の設定
32:     w1.line ( t*l );                                     //  変換して描画
33:
34:
35: //  Ｙ字型図形の回転移動
36:     Win3D   w2("w2:回転移動");                           //  ウィンドウ生成
37:     w2.axis ();                                          //  座標軸の表示
38:     w2.color( CYAN );                                    //  描画色の設定
39:     w2.line ( l );                                       //  Ｙ字形の描画
40:
41:     float   a=90*PIR;
42:     t=TMatrix(  cos(a), -sin(a),0,      0,               //  変換行列の設定
43:                 sin(a), cos(a), 0,      0,               //  Z 軸回転（90 度）
44:                 0,      0,      1,      0,
45:                 0,      0,      0,      1   );
46:     w2.color( RED );                                     //  描画色の設定
47:     w2.line ( t*l );                                     //  変換して描画
48:
49:     a=-45*PIR;
50:     t=TMatrix(  1,      0,      0,      0,               //  変換行列の設定
```

```
                0,      cos(a), -sin(a), 0,              //  X軸回転 (-45度)
                0,      sin(a),  cos(a), 0,
                0,      0,       0,      1  );
    w2.color( MAGENTA );                                 //  描画色の設定
    w2.line ( t*l );                                     //  変換して描画

    pause("EX3_6");
}
```

リスト3-7　EX3_7.cpp

```
 1:          3次元空間の図形の変換（座標の演算を含む）
 2:
 3: #include        "graph2.h"
 4:
 5: main()
 6: {
 7:     TMatrix t;                                        //  変換行列
 8:     Line    l=Line( Point(0,50,0),    Point(23,27,0), //  Y字形の頂点座標の設定
 9:                     Point(23,0,0),    Point(33,0,0),
10:                     Point(33,27,0),   Point(50,50,0),
11:                     Point(40,50,0),   Point(30,36,0),
12:                     Point(16,50,0),   Point(0,50,0) );
13:
14: //  Y字型図形のスケーリング
15:     Win3D   w1("w1:スケーリング");                    //  ウィンドウ生成
16:     w1.axis ();                                       //  座標軸の表示
17:     w1.color( CYAN );                                 //  描画色の設定
18:     w1.line ( l );                                    //  Y字形の描画
19:
20:     t=TMatrix(  3,      0,      0,      0,            //  変換行列の設定
21:                 0,      3,      0,      0,            //  各方向を3倍
22:                 0,      0,      3,      0,
23:                 0,      0,      0,      1  );
24:     w1.color( RED );                                  //  描画色の設定
25:     w1.line ( t*l );                                  //  変換して描画
26:
27:     t=TMatrix(  0.5,    0,      0,      0,            //  変換行列の設定
28:                 0,      2,      0,      0,            //  X方向に0.5倍
29:                 0,      0,      1,      0,            //  Y方向に2倍
30:                 0,      0,      0,      1  );
31:     w1.color( MAGENTA );                              //  描画色の設定
32:     w1.line ( t*l );                                  //  変換して描画
33:
34:
35: //  Y字型図形の対象変換
36:     Win3D   w2("w2:対象変換");                        //  ウィンドウ生成
37:     w2.axis ();                                       //  座標軸の表示
38:     w2.color( CYAN );                                 //  描画色の設定
39:     w2.line ( l );                                    //  Y字形の描画
40:
41:     t=TMatrix(  -1,     0,      0,      0,            //  変換行列の設定
42:                 0,      1,      0,      0,            //  YZ平面対象
43:                 0,      0,      1,      0,
44:                 0,      0,      0,      1  );
45:     w2.color( RED );                                  //  描画色の設定
46:     w2.line ( t*l );                                  //  変換して描画
47:
48:     t=TMatrix(  -1,     0,      0,      0,            //  変換行列の設定
49:                 0,      -1,     0,      0,            //  Z軸対象
50:                 0,      0,      1,      0,
51:                 0,      0,      0,      1  );
52:     w2.color( MAGENTA );                              //  描画色の設定
53:     w2.line ( t*l );                                  //  変換して描画
54:
55:     pause("EX3_7");
```

```
56: }
```

リスト3-8　EX3_8.cpp

```
 1: //      原点を通る直線を中心とする回転
 2:
 3: #include        "graph2.h"
 4:
 5: main()
 6: {
 7:     TMatrix t;                                                   //  変換行列
 8:     Line    l=Line( Point(0,50,0),      Point(23,27,0),  //  Ｙ字形の頂点座標の設定
 9:                     Point(23,0,0),      Point(33,0,0),
10:                     Point(33,27,0),     Point(50,50,0),
11:                     Point(40,50,0),     Point(30,36,0),
12:                     Point(16,50,0),     Point(0,50,0) );
13:
14:     Win3D   w1("EX3_8");                                     //  ウィンドウ生成
15:     w1.axis ();                                              //  座標軸の表示
16:     w1.color( CYAN );                                        //  描画色の設定
17:     l=move(Point(50,50,0))*l;
18:     w1.line ( l );                                           //  Ｙ字形の描画
19:
20:     Vector  a=unit( Vector(2,1,0) );                         //  回転軸の方向
21:
22:     w1.color( BLUE );
23:     w1.line ( a*0, a*200 );                                  //  回転軸の描画
24:
25:     Vector  axy=unit( Vector(a.x,a.y,0) );
26:     TMatrix t1=rot_z(-axy.y, axy.x);
27:     TMatrix t5=rot_z( axy.y, axy.x);
28:     Vector  axz=t1*unit(a);
29:
30:     TMatrix t2=rot_y( axz.z, axz.x);
31:     TMatrix t4=rot_y(-axz.z, axz.x);
32:
33:     w1.color( RED );                                         //  描画色の設定
34:     TMatrix t3=rot_x(30);                                    //  回転角
35:     t=t5*t4*t3*t2*t1;                                        //  変換行列の合成
36:     w1.line ( t*l );                                         //  変換して描画
37:
38:     t3=rot_x(60);                                            //  回転角
39:     t=t5*t4*t3*t2*t1;                                        //  変換行列の合成
40:     w1.line ( t*l );                                         //  変換して描画
41:
42:     pause("EX3_8");
43: }
```

# 4．投影法

この章では，３次元空間の図形（点，線）が任意の位置(視点)からどのように見えるのかについて考えます．すなわち，視点に近いものは大きく見え，また視点から遠いものは小さく見えるというような遠近感です．なお，本章で説明する変換は，すでに前章のWin3Dクラスを用いた描画で使用しています．

## ４．１　投影変換の種類と投影図形

投影とは，ものの影に例えることができます．投影には図4-1および図4-2に示すように，中心投影と平行投影があります．図4-1をたとえると，投影中心から光を発している場合，三角形ＡＢＣによって$XY$平面にできる影が投影図形に相当します．図4-2は，投影中心が$Z$軸方向で無限遠方の場合です．影ができる平面（図4-1および図4-2では$XY$平面）を投影面といいます．

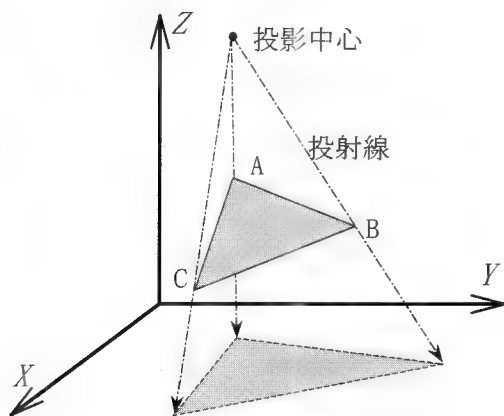
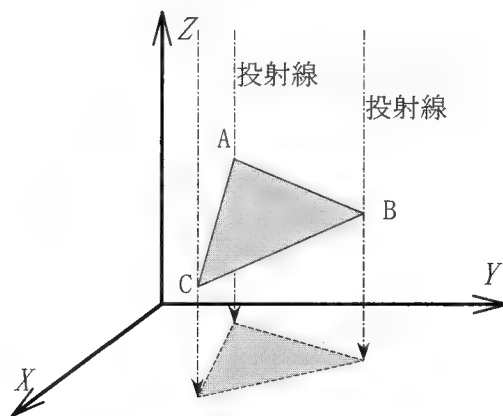



図4-1　中心投影　　図4-2　平行投影

## ４．２　平行投影

（１）　投影方向が$Z$軸方向の場合

まず，簡単な平行投影の場合について考えてみましょう．座標変換の場合と同様に，図形を構成する線分は投影面でも線分になります．したがって，図形が線分で構成されている場合，その線分の両端の点の投影座標を求めることによって，投影図形を構成する線分を求めることができます．

図4-2の場合，投影方向が$Z$軸方向なので，図形を構成する点（$x,y,z$）が投影される$XY$平面上の座標（$x',y',0$）は，次式によって求めることができます．これは$X,Y$座標そのものであり，単に$Z$座標がなくなっただけになります．そのため，投影面が$XY$平面に平行な平面であ

れば，投影図形は同一なものとなります．

$$x' = x \\ y' = y \tag{4-1}$$

（2）　投影方向が任意方向の場合

次に，図4-3のような原点を含む任意の平面への投影について考えましょう．図4-4に示すように投影面の法線ベクトル（投影面に垂直で$U$軸方向のベクトル）を$N$とし，仰角$e$と方位角$a$を用いて式(4-2)で表します．投影方向は法線ベクトル$N$方向とします．ここで，図4-3に示すように新たな座標軸$UVW$を考えましょう．投影面上の座標軸を$U$および$V$とし，$Z$軸の投影が$V$軸に重なるようにします．そして，$UVW$軸は互いに直交するようにします（$W$軸は法線ベクトル$N$と平行になる）．

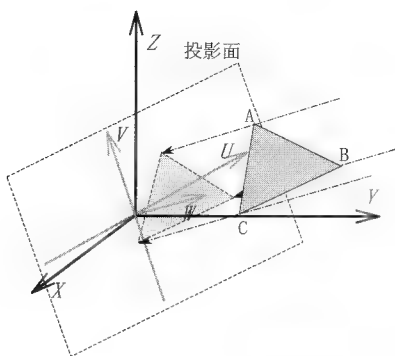
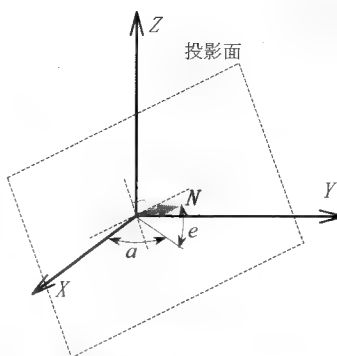
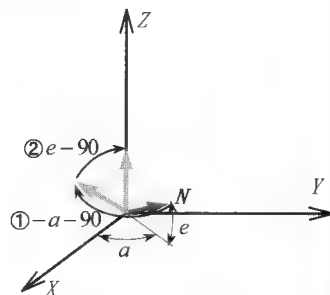



**図4-3**　任意方向への平行投射　**図4-4**　投影面の法線ベクトル　**図4-5**　法線ベクトルの回転

$$N = (\cos a \cdot \cos e, \sin a \cdot \cos e, \sin e) \tag{4-2}$$

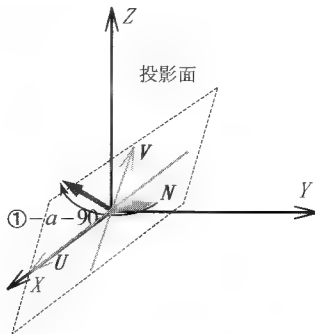
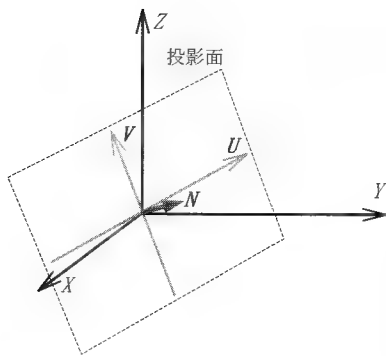



**図4-6**　回転の様子

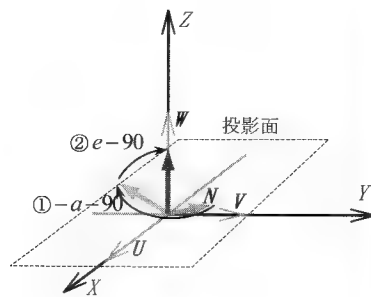



図4-2と図4-3を比較してみましょう．図4-3の座標軸を$XYZ$軸から$UVW$軸に置き換えると，図4-2のようになり，投影座標を容易に求められることに気がつきます．図4-5に示すように，法線ベクトル$N$を以下のように回転することによって，$Z$軸に重ねることができます．それは，同時に$UVW$軸がそれぞれ$XYZ$軸に重なることになります．その様子を図4-6に示します．

① $Z$軸を中心に$-a-90$度回転する(変換行列はrot_z(−a−90)で求められる)．
　$U$軸が$X$軸に重なる．
② $X$軸を中心に$e-90$度回転する（変換行列はrot_x(e−90)で求められる）．
　$UVW$軸がそれぞれ$XYZ$軸に重なる．

したがって，図形の座標$p=(x,y,z)$から次式により，上記の変換座標$p'=(x',y',z')$を得ることができます．そして，投影後の座標は，変換座標$p'$の$X$および$Y$座標値となります．

$$p'=\mathrm{rot_}x(e-90)*\mathrm{rot_}z(-a-90)\cdot p \tag{4-3}$$

いいかえると，$XYZ$座標（$XYZ$軸を使用）で表されている図形の座標を，$UVW$座標（$UVW$軸を使用）で表したことに相当します．ヘッダファイルgraph2.hに記述されているWin3Dクラスのsetview関数で，上記の変換行列を計算しています．

## 4．3　中心投影

（1）　投影方向が$Z$軸方向の場合

まず，図4-7に示すような投影中心$(0,0,z_0)$が$Z$軸上にあり，$XY$平面への投影について考えます．平行投影の場合と同様に，図形を構成する線分は投影面でも線分になります．したがって，図形が線分で構成されている場合，その線分の両端の点の投影座標を求めることによって，投影図形を構成する線分を求めることができます．

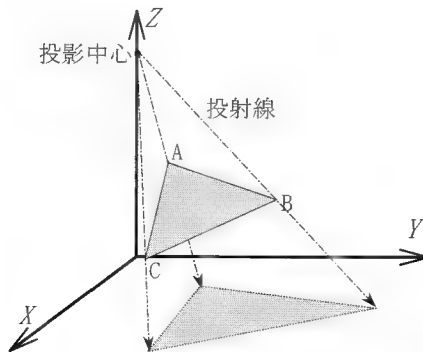
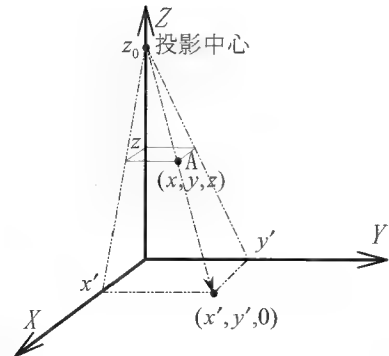



**図4-7**　中心投影　　**図4-8**　図形の座標と投影座標の関係

図4-8に，図形の座標$(x,y,z)$と投影座標$(x',y',0)$の関係を示します．この関係から式(4-4)によって，投影座標を求めることができます．

$$\begin{aligned} x' &= x\cdot\frac{z_0}{z_0-z}=\frac{x}{1-z/z_0} \\ y' &= y\cdot\frac{z_0}{z_0-z}=\frac{y}{1-z/z_0} \end{aligned} \tag{4-4}$$

投影座標は，図形の $X, Y$ 座標の値をそれぞれ $1/(1-z/z_0)$ 倍したものとなり，$Z$ 座標の値によって倍率が変化します．このことによって，$Z$ 座標の値が投影中心に近い場合は大きくなり，遠い場合は小さくなります．すなわち，遠近感がでます．本書では，投影面は $XY$ 平面とすることとし，式(4-4)を用います．式(4-4)は，次式のように行列を用いた式(4-5)および式(4-6)で計算することができます．この変換のために，ヘッダファイルmatrix.hのperspect関数で式(4-5)の変換行列を生成しています．また，ヘッダファイルgraph2.hのscreen関数で式(4-5)および式(4-6)で計算を行って，スクリーン座標に変換しています．

$$\begin{bmatrix} x'' \\ y'' \\ z'' \\ w'' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & -1/z_0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{4-5}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x''/w'' \\ y''/w'' \\ z'' \\ 1 \end{bmatrix} \tag{4-6}$$

スクリーンに描画する場合には，変換後の $Z$ 座標の値は関係ありませんが，変換後のスクリーン座標では投影中心が原点となるように，$Z$ 座標の値は次のようにしています．投影中心では0，投影中心から離れるにしたがって小さくなり投影面上では $-z_0$，それ以降は値が小さくなっていきます．本書では，数学の幾何学で用いている座標系(右手系)を使用していますが，変換後の $Z$ 座標の値が正となるようにするため $Z$ 軸の正負を反転した左手系の座標軸を用いる方法もあります．

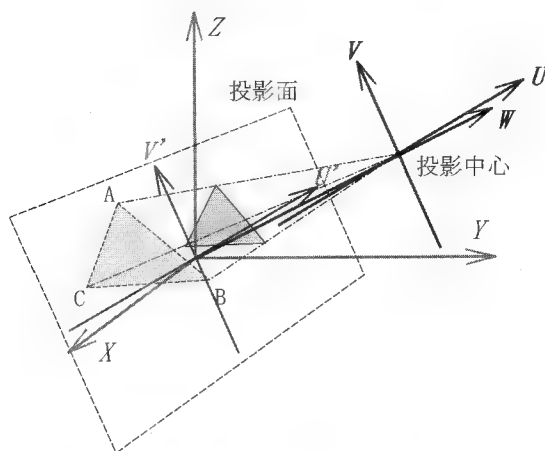



図4-9　任意方向への中心投射

### (2)　投影方向が任意方向の場合

図4-9に示すように，投影方向が任意方向の場合について考えます．平行投影の場合と同様に，座標軸を置き換えた後に(1)の方法で投影座標を求めることができます．したがって，式(4-3)の座標変換を行い，次に式(4-5)および式(4-6)の計算を行うことによって投影座標を求めることができます．

## 4．4　透視変換とビューイング変換

図4-10に示すように立方体を視点から眺めた場合，どのような図形が見えるのでしょうか．図4-9の中心投影と比較してみると，視点は投影中心に対応しており，スクリーン(画面)は投影面に対応しています．また，対象となる図形が三角形１つから正方形６つに増えていると考えることができます．

このように投影した場合を，透視変換または透視投影といいます．透視変換では，遠近感は出ますが，図4-10に示すように重なりによって本来見えない部分も表示されてしまいます．本来見えない部分を表示しないようにする処理を隠面処理または隠線処理といいます．

透視変換および隠線処理を行い，視点から眺めた場合の視界を求めることをビューイング変換または視野変換といいます．図4-11に示します．

隠面処理および物体の表現方法については，次章で考えることにします．

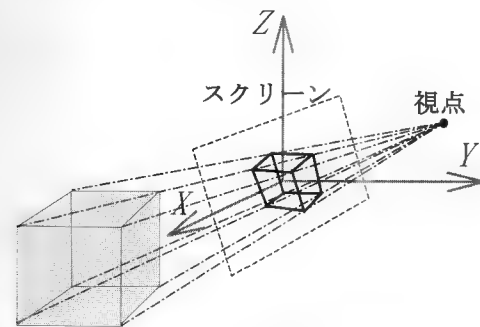



図4-10　透視変換

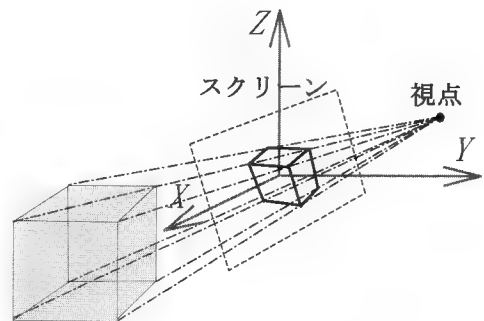



図4-11　ビューイング変換

３次元表示用クラスのWin3Dクラスでは，視点位置を設定するsetview関数で透視変換のための変換行列$t$を計算しています．そして，screen関数によって式(4-5)および式(4-6)に示す投影座標を求める計算を行っています．ここで，投影面(スクリーン)には，原点付近は拡大・縮小されずにそのままの大きさで表示されるようにしています．

graph2.hでのビューイング変換関連の関数

```
void origin(int x, int y);                          原点位置(スクリーン座標)の指定
void    setview(float a, float e, float d=1000);  視点方向(a:方位角  e:仰角  d:距離)
void    setview(Point& n);                        視点位置(n:視点座標)
Point  screen(Point& p);                          スクリーン上の座標を返す
```

視点との距離を変えた場合（方位角30度，仰角30度）のプログラム例(EX4_1.cpp)を，リスト4-1に実行結果を図4-12に示します．

ウィンドウw1:　視点は10000遠方

ウィンドウw2:　視点は500遠方

ウィンドウw3:　視点は300遠方

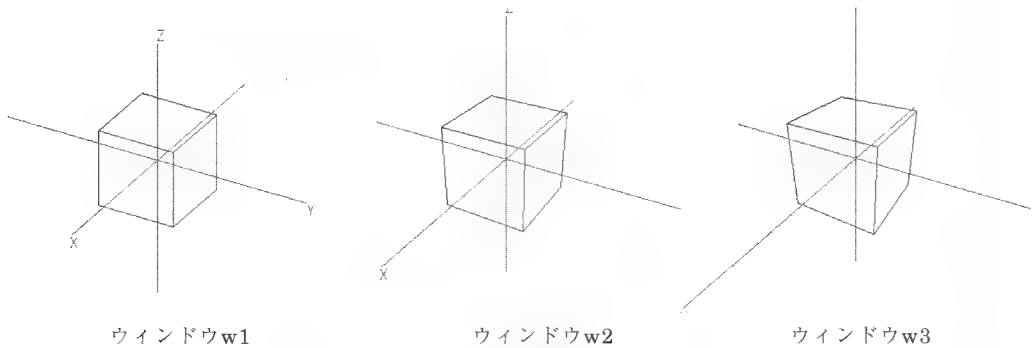



図4-12　視野変換(EX4_1.cpp)

視点の方向を変えた場合（距離10000）のプログラム例(EX4_2.cpp)を，リスト4-2に実行結果を図4-13に示します．

ウィンドウw1:　視点は方位角30度，仰角30度

ウィンドウw2:　視点は方位角60度，仰角30度

ウィンドウw3:　視点は方位角30度，仰角60度

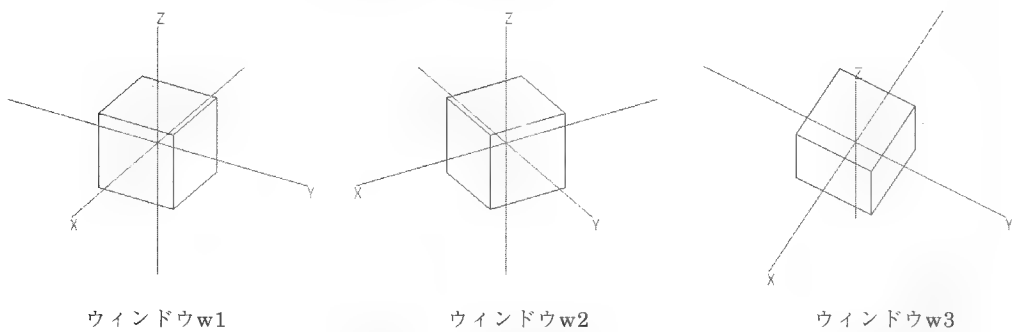



図4-13　視野変換(EX4_2.cpp)

リスト 4-1　EX4_1.cpp

```
 1: //      3次元空間に簡単な図形を画く (視点からの距離を変化)
 2:
 3: #include        "graph2.h"
 4:
 5: main()
 6: {
 7:     Line    l1=Line(Point(-1, 1, 1),Point( 1, 1, 1),
 8:                     Point( 1, 1,-1),Point(-1, 1,-1),
 9:                     Point(-1, 1, 1),Point(-1,-1, 1) );
10:     Line    l2=Line(Point(-1,-1, 1),Point( 1,-1, 1),
11:                     Point( 1, 1, 1),Point( 1, 1,-1),
12:                     Point( 1,-1,-1),Point( 1,-1, 1) );
13:
14:     // 方位角30度、仰角30度、10000遠方
15:     Win3D   w1("w1:10000遠方から");                     // ウィンドウ生成
16:     w1.setview(30,30,10000);                            // 視点の設定
17:     w1.axis ();                                         // 座標軸の表示
18:     w1.color( RED );                                    // 描画色の設定
19:     w1.line ( 50*l1 );
20:     w1.line ( 50*l2 );
21:
22:     // 方位角30度、仰角30度、500遠方
23:     Win3D   w2("w2:500遠方から");                       // ウィンドウ生成
24:     w2.setview(30,30,500);                              // 視点の設定
25:     w2.axis ();                                         // 座標軸の表示
26:     w2.color( RED );                                    // 描画色の設定
27:     w2.line ( 50*l1 );
28:     w2.line ( 50*l2 );
29:
30:     // 方位角30度、仰角30度、300遠方
31:     Win3D   w3("w3:300遠方から");                       // ウィンドウ生成
32:     w3.setview(30,30,300);                              // 視点の設定
33:     w3.axis ();                                         // 座標軸の表示
34:     w3.color( RED );                                    // 描画色の設定
35:     w3.line ( 50*l1 );
36:     w3.line ( 50*l2 );
37:
38:     pause("EX4_1");
39: }
```

リスト 4-2　EX4_2.cpp

```
 1: //      3次元空間に簡単な図形を画く (視点の方向を変化)
 2:
 3: #include        "graph2.h"
 4:
 5: main()
 6: {
 7:     Line    l1=Line(Point(-1, 1, 1),Point( 1, 1, 1),
 8:                     Point( 1, 1,-1),Point(-1, 1,-1),
 9:                     Point(-1, 1, 1),Point(-1,-1, 1) );
10:     Line    l2=Line(Point(-1,-1, 1),Point( 1,-1, 1),
11:                     Point( 1, 1, 1),Point( 1, 1,-1),
12:                     Point( 1,-1,-1),Point( 1,-1, 1) );
13:
14:     // 方位角30度、仰角30度、10000遠方
15:     Win3D   w1("w1:方位角30度、仰角30度");              // ウィンドウ生成
16:     w1.setview(30,30,10000);                            // 視点の設定
17:     w1.axis ();                                         // 座標軸の表示
18:     w1.color( RED );                                    // 描画色の設定
19:     w1.line ( 50*l1 );
20:     w1.line ( 50*l2 );
21:
```

```
22:     // 方位角60度、仰角30度、10000遠方
23:     Win3D   w2("w2:方位角60度、仰角30度");              // ウィンドウ生成
24:     w2.setview(60,30,10000);                          // 視点の設定
25:     w2.axis ();                                       // 座標軸の表示
26:     w2.color( RED );                                  // 描画色の設定
27:     w2.line ( 50*l1 );
28:     w2.line ( 50*l2 );
29:
30:     // 方位角30度、仰角60度、10000遠方
31:     Win3D   w3("w3:方位角30度、仰角60度");              // ウィンドウ生成
32:     w3.setview(30,60,10000);                          // 視点の設定
33:     w3.axis ();                                       // 座標軸の表示
34:     w3.color( RED );                                  // 描画色の設定
35:     w3.line ( 50*l1 );
36:     w3.line ( 50*l2 );
37:
38:     pause("EX4_2");
39: }
```

# 5．形状モデルと隠面処理

この章では，３次元空間での立体の形状モデルの種類について説明します．そして，形状モデルのうちのポリゴンモデル（後述）について，描画するための方法（隠面処理の方法）を考えていきます．

## 5.1 形状モデル

ＣＧ画像を作成する手順は，形状の作成と描画の２つに分けられます．立体の形状を作ることをモデリング(modeling)といい，描画することをレンダリング(rendering)といいます．前者の立体形状を表すモデルとして，以下の３つがあります．

（１） ワイヤーフレームモデル

稜線の集合によって物体を記述するモデルです．例えると，直線や曲線できた針金細工のようなものになり，このような表示をフイヤーフレームと呼びます（図5-3, 図5-5参照）．

（２） ポリゴンモデル（サーフェイスモデル）

ポリゴン（多角形）の集合（多面体）によって物体を記述するモデルです．例えると，紙でできた立体模型（立方体，多面体など）のようなものであり，隠面処理および各面の明るさなどを表示することができます．高速な描画が可能なので，ゲームなどでよく使われています．ワイヤーフレームとしても表示することができます（図5-6参照）．

（３） ソリッドモデル

中身がつまったものとして物体を記述するモデルです．レイトレーシング法（後述）では，数学的な関数を用いて記述され，鏡のような反射やガラスのような屈折を含む非常にリアルな表示が可能となります．しかし，ポリゴンモデルに対して，描画時間がかかります．ソリッドモデルについては，７章で説明します（図7-14, 図7-18参照）．

## 5.2 ポリゴンモデルの記述方法

本章では，これ以降ポリゴンモデル（サーフェイスモデル）について考えていきます．まず，ポリゴンモデルの記述方法について説明します．ポリゴンモデルの構成要素であるポリゴン（同一平面上の多角形）を記述するために，Polygonクラスを用意しています．Polygonクラスは多角形の頂点の集合であり，Lineクラスとよく似ていますが，各点は同一平面上にあり始点と終点が一致します．Polygonクラスは，ヘッダファイルpolygon.h(章末に示す)に記述されており，ヘッダファイルgraph3.h(後述)にインクルードされています（ヘッダファイルgraph3.hは，ヘッダファイルgraph2.hを拡張したもので，ポリゴンモデルやソリッドモデルを扱えるようにしています）．以下の記述のように，PointクラスおよびLineクラスのデータからPolygonクラスのデータを生成することができます（三角形のポリゴンの例）．

```
Polygon   a=Polygon(Point(10,0,0),Point(10,10,0),Point(0,10,0));
Line      l=Line(Point(10,0,0),Point(10,10,0),Point(0,10,0));
Polygon   a=Polygon(l);
```

さらに，ポリゴンモデル(多面体)を記述するためにSurfaceクラスを用意しています．Surfaceクラスはポリゴンの集合であり，Lineクラスのデータからrevolve_z,sweep_xz関数によって生成することができるようにしています（Surfaceクラスはデータ構造およびその処理を単純にするために単なるポリゴンの集合として記述していますが，データ構造を工夫することによって必要な記憶容量および座標変換などの際の計算量を削減することができます）．

円弧を近似したLineクラスのデータを生成するcircle_xy,circle_yz,circle_xz,関数を用意しています．Lineクラスのデータを使用して，revolve_z関数によって回転体（円すい（多角すい），円柱（多角柱），球体,トーラス（ドーナツ状））のSurfaceクラスのデータを生成することができます．また，sweep_xz関数によって，線分$a$を線分$b$に沿って移動した軌跡を形状とした形状データを生成することができます（図5-2参照）．Surfaceクラスは，ヘッダファイルsurface.h（巻末に示す）に記述されています．

```
Line      circle_xy(int n, float b=0, float e=360);
Line      circle_yz(int n, float b=0, float e=360);
Line      circle_xz(int n, float b=0, float e=360);
```

平面上に原点を中心に半径１の円の円周上の点（角度がb->e）を連結したｎ本の線分を生成する．回転方向はそれぞれZ,X,Y軸回転移動の場合と同じである．

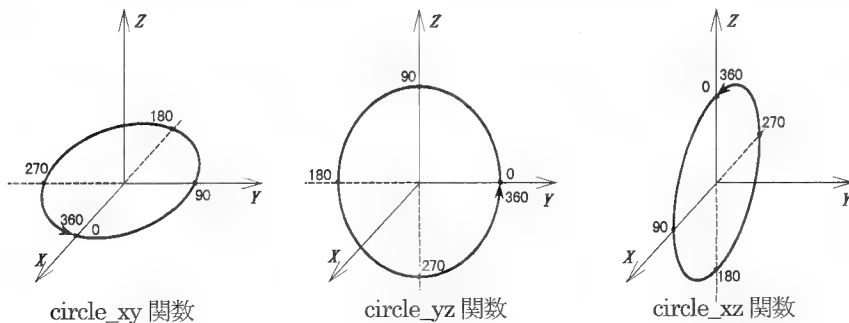



図5-1　関数による円弧

```
Surface   revolve_z(Line& a, int n , float b=0, float e=360);
```

線分aをZ軸回転（角度がb->e,n分割）しながら，ポリゴンモデルを生成する．

```
Surface   sweep_xz(Line& a, Line& b);
```

XZ平面の線分aを線分bに沿って移動した軌跡のポリゴンモデルを生成する．

トーラスの一部の生成例(図5-2参照)

```
Line      a= move(100,0,0)*50*circle_xz(8);      // 半径50の円(8角形)の　生成
Surface f= revolve_z( a, 3, 0, 90);              // トーラスの生成
```

曲がったパイプの生成例(図5-2参照)

```
    Line      a= 50*circle_xz(8);                // 半径50の円(8角形)の生成
```

```
Line    b= Line( Point(0,0,0), Point(20,20,0),
              Point(20,40,-10), Point(10,60,-10)); // 線分の生成
Surface f= sweep_xz( a, b );                       // パイプの生成
```

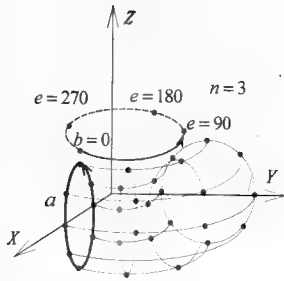
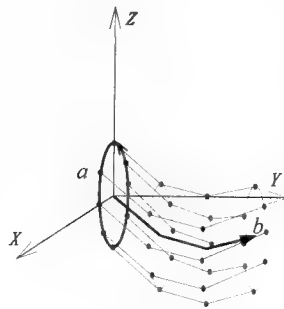



図5-2　関数による立体生成

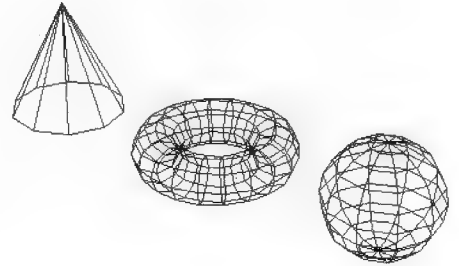



図5-3　立体モデルの生成例(EX5_1.cpp)

PolygonクラスおよびSurfaceクラスはLineクラスと同様に，座標変換（変換行列との積）ができ，移動や拡大縮小ができます(詳細は付録を参照して下さい).

立体モデルの生成例（球体，トーラス，円すい）として，生成したモデルをワイヤーフレーム表示するプログラム例（EX5_1.cpp）をリスト5-1に，実行結果を図5-3に示します．PolygonクラスおよびSurfaceクラスを描画するために，Win3Dクラスを拡張して，ヘッダファイルgraph3.h(巻末に示す)に記述しています．graph3.hには，Polygon.hとSurface.hがインクルードされています．

## 5.3　法線ベクトル法による隠面処理

図5-3はワイヤーフレーム表示なので，視点から見えない部分(立体の裏側)まで表示されています．見えない部分を，描画しないようにすることを隠面処理といいます．ポリゴンモデルに対する隠面処理の方法として，立体を構成するポリゴンの法線ベクトルを利用する方法があり，この方法を法線ベクトル法といいます．視点から立体を構成するポリゴンを見て，立体の外側の面が見える場合には描画し，内側の面が見える場合には描画しないようにします．外側の面か内側の面かの判定に，ポリゴンの法線ベクトルを使用します．立体を構成するポリゴンの法線ベクトルは，立体の外側を向いているものとします．図5-4に示すように，ポリゴンの法線ベクトルを$N$，視線の方向ベクトルを$V$とすると，その内積は次のような意味を持ちます（ベクトルの演算については付録を参照）．

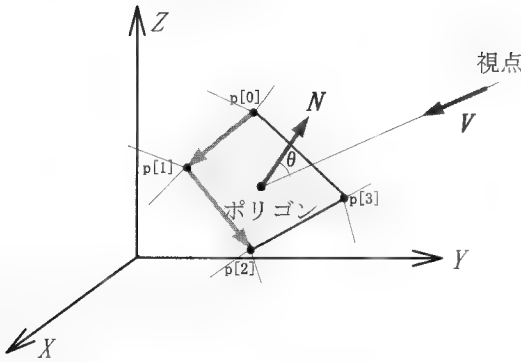



図5-4　ポリゴンと法線ベクトル

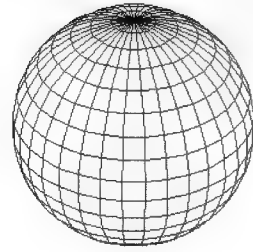



図5-5　法線ベクトル法(EX5_2.cpp)

$$N \cdot VP = |N| \cdot |VP| \cdot \cos\theta \\ = N_x \cdot VP_x + N_y \cdot VP_y + N_z \cdot VP_z \tag{5-1}$$

・外側の面が見える場合（$0 \leq \theta \leq \pi/2$），$\cos(\pi-\theta) \leq 0$となり，内積の値も0以下となる．
・内側の面が見える場合（$\pi/2 < \theta \leq \pi$），$\cos(\pi-\theta) > 0$となり，内積の値も正となる．

（$N_x, N_y, N_z$は，$N$の$X,Y,Z$座標または$X,Y,Z$成分を表すものとする）

したがって，内積の値が0以下の場合には，そのポリゴンを描画します．

前項のsweep_xz,revolve_z関数は，図5-4のようなポリゴンでSurfaceクラスのデータを生成します．ポリゴンを記述している点配列$p$を用いて，次式で法線ベクトル$N$（単位ベクトルとは限らない）を計算することができます．

$$\begin{aligned} A &= p[0]-p[2] & N_x &= A_y \cdot B_z - A_z \cdot B_y \\ B &= p[1]-p[3] & N_y &= A_z \cdot B_x - A_x \cdot B_z \\ N &= A \times B & N_z &= A_x \cdot B_y - A_y \cdot B_x \end{aligned} \tag{5-2}$$

法線ベクトル$N$は，頂点間をベクトルとした２つのベクトル$A$と$B$の外積（プログラム中では外積を"%"と記述している）によって計算することができます．ベクトル$N$は，ベクトル$A$とベクトル$B$に直交するベクトルとなります．外積の計算を行うことにより，すべてのポリゴンで外側に向いた法線ベクトル$N$を求めることができます(外積については，付録を参照)．

法線ベクトル法を用いて，球体の隠面処理(EX5_2.cpp)を行うプログラム例をリスト5-2に，実行結果を図5-5に示します．

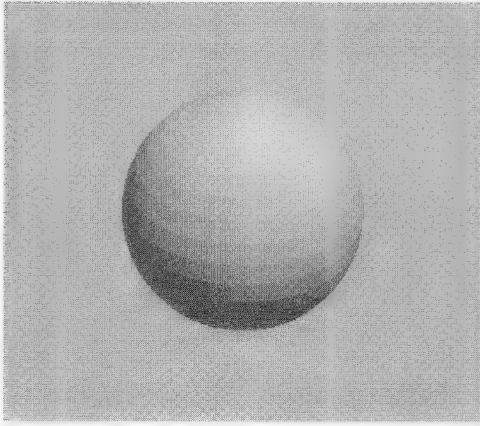



図5-6　法線ベクトル法(EX5_3.cpp)

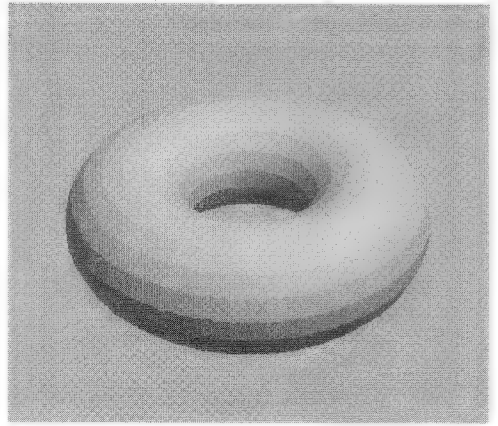



図5-7　Zソート法(EX5_4.cpp)

いよいよ立体的に見えてきましたので，EX5_2.cppを変更して明暗を付けることにします．そのプログラム例(EX5_3.cpp)をリスト5-3に，実行結果を図5-6に示します．プログラム中で，

```
w.color( (max(-N*L,.0)*0.8+0.2)*c );
```

の行で各ポリゴンに明暗を付けています．明暗の付け方は次章で説明します．

## 5.4　Zソート法による隠面処理

法線ベクトル法では，球体のような単純な物体の場合にはうまく隠面処理ができますが，複数の物体が重なって見える場合や複雑な物体ではうまく隠面処理ができません．それは，物体を構成するポリゴンのうちで外側の面が見えるポリゴンを描画しており，このようなポリゴンが重なって見える場合についての考慮をしていないためです．

前述の問題点は，Zソート法を用いることによって解決することができます．Zソート法では，物体を構成するポリゴンを視点から遠い順に描画（塗りつぶす）します．すなわち，透視変換後のポリゴンの前後関係にしたがってZ座標の小さい方（遠方）からポリゴンの内部を塗りつぶしていきます．トーラス（ドーナツ形状）について，法線ベクトル法を適用したプログラム例(EX5_4.cpp)をリスト5-4に，実行結果を図5-7に示します．

## 5.5　Zバッファ法による隠面処理

Zソート法では，ポリゴン単位で前後関係を考えていましたが，図5-8に示すように物体が交わり，ポリゴンが交差する場合にはうまく隠面処理ができません．これを解決する方法が，Zバッファ法です．この方法では，ポリゴンを描画する際に，画素毎に前後関係を考えます．そのため，スクリーン（画面）全体の画素に対する奥行き（透視変換後のZ座標）を記録しておくためのZバッファを用意しておき，十分小さな値に初期化しておきます．そして，ポリゴンを描画する際に画素毎に，透視変換後のZ座標の値（小さいほど視点から遠い）とその画素の位置に対応するZバッファの値を比較して，Z座標の値が大きい場合（視点に近い）に描画を行い，Zバッファ

をZ座標の値に更新します．Z座標の値が小さい場合には，何もしません．この処理によって，スクリーンの各画素に対応するZバッファには，その画素に描画したポリゴン(ポリゴンの画素)のうちの視点に最も近いZ座標の値が保持されることになります．

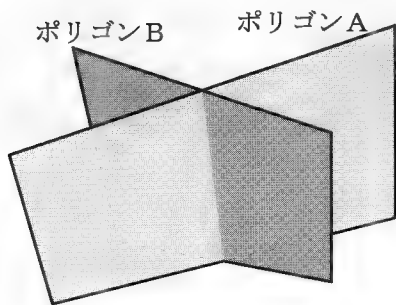



図5-8　ポリゴンが交差する場合

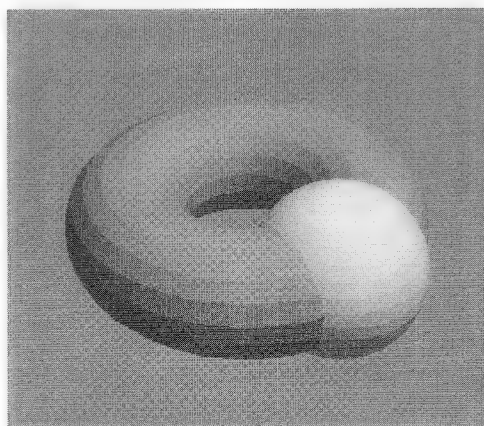



図5-9　Zバッファ法(EX5_5.cpp)

トーラスと球体が交わる場合について，Zバッファ法を適用したプログラム例(EX5_5.cpp)をリスト5-5に，実行結果を図5-9に示します．

プログラムでは，paint関数でポリゴンを描画しています．paint関数では，図5-10に示すようにポリゴン内部の点 P=$(x,y)$を，以下に示す方法で描画していきます．

① ポリゴンの各頂点を，screen関数によってスクリーン座標($P_0 \sim P_3$)に変換する．

② 変換後の各頂点の$Y$座標($P_{0y} \sim P_{3y}$)の範囲（$y$_min～$y$_max）を求める．

③ 各$Y$座標（$y$_min～$y$_max）での範囲$P_4$(左端)～$P_5$(右端)を求める．

$P_4, P_5$は下式で計算する．プログラムではsup関数で計算している．

④ $P_4 \sim P_5$($P_{4x} \sim P_{5x,y}$)までの各$P=(x,y)$を，Zバッファ法で描画する．

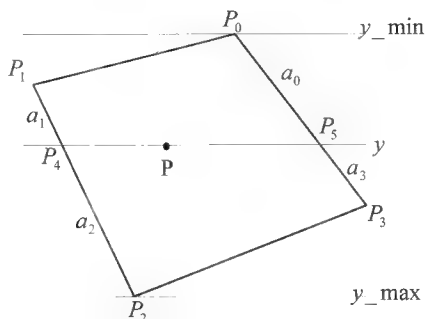



図5-10　ポリゴンの描画

$$
\begin{aligned}
a_0 &= P_y - P_{0y} \\
a_1 &= P_y - P_{1y} \\
a_2 &= P_{2y} - P_y \\
a_3 &= P_{3y} - P_y \\
a_4 &= P_x - P_{4x} \\
a_5 &= P_{5x} - P_x
\end{aligned}
\qquad
\begin{aligned}
P_4 &= \frac{a_2 P_1 + a_1 P_2}{a_1 + a_2} \\
P_5 &= \frac{a_3 P_0 + a_0 P_3}{a_0 + a_3} \\
P &= \frac{a_5 P_4 + a_4 P_5}{a_4 + a_5}
\end{aligned}
\tag{5-3}
$$

## 演習問題

5-1 プログラム例EX5_3.cppを変更して，半分の球体を描画せよ（不自然な点を観察せよ）．
（main内を，以下のように変更する）

```
Surface  f=revolve_z( 100*a, 18, 90, 270 );
```

5-2 プログラム例EX5_4.cppを変更して，プログラム例EX5_5.cppのようなトーラスと球体が重なるようすを，$Z$ソートで描画せよ．$Z$ソートでは，うまく描画できないことを確認せよ．

リスト 5-1　EX5_1.cpp

```
 1: //      円錐、球体、トーラス（ワイヤーフレーム）を描画する
 2: //      Surface クラスを使用する
 3:
 4: #include        "graph3.h"
 5:
 6: main()
 7: {
 8:     Win3D   w("EX5_1",WHITE);
 9:     w.color( BLACK );
10:
11:     Line    a= Line(Point(50,0,0),Point(0,0,100));      // 円錐の素（直線）の生成
12:     Surface f= revolve_z( a, 10 );                       // 円錐（ポリゴン集合)の生成
13:     w.line ( move(Point(0,-150,0))*f );
14:
15:     a= circle_xz( 10, 0, 180);                          // 球体の素（半円）の生成
16:     f= revolve_z( 50*a, 10 );                            // 球体（ポリゴン集合）の生成
17:     w.line( move(0,150,0)*f );
18:
19:     a= circle_xz( 8);                                    // トーラスの素（円）の生成
20:     f= revolve_z( move(50,0,0)*20*a, 20 );               // トーラス（ポリゴン集合）の生成
21:     w.line( f );
22:
23:     pause("EX5_1");
24: }
```

リスト 5-2　EX5_2.cpp

```
 1: //      法線ベクトル法を利用した陰線（陰面）処理
 2: //      球体を描画する
 3:
 4: #include        "graph3.h"
 5:
 6: main()
 7: {
 8:     Line    a= circle_xz( 18, 0, 180);                  // 球体の素（半円）の生成
 9:     Surface f= revolve_z( 100*a, 36 );                   // 球体（ポリゴン集合）の生成
10:
11:     Win3D   w("EX5_2",WHITE);
12:     Point   V0=Point(500,500,500);                      // 視点の記述（座標：V0)
13:     w.setview(V0);
14:     w.color(RED);
15:
16:     // 球体（ポリゴン集合）の描画
17:     Vector  V,N;
18:     for(int i=0; i<f.n; i++){
19:         Polygon p=f.p[i];
20:         V=unit(p.p[0]-V0);                               // 視線ベクトル
21:         N=unit((p.p[0]-p.p[2])%(p.p[1]-p.p[3]));         // 法線ベクトルの計算（外積）
22:         if( N*V <=0 )w.line( p);                         // ポリゴン枠の描画
23:     }
24:
25:     pause("EX5_2");
26: }
```

リスト 5-3　EX5_3.cpp

```
 1: //      法線ベクトル法を利用した陰線（陰面）処理
 2: //      球体を描画する
 3:
 4: #include        "graph3.h"
 5:
 6: main()
```

```
 8:     Line    a= circle_xz( 18, 0, 180);                 //  球体の素（半円）の生成
 9:     Surface f= revolve_z( 100*a, 36 );                 //  球体（ポリゴン集合）の生成
10:     Color   c= RED;
11:
12:     Win3D   w("EX5_3", WHITE*0.3+GREEN*0.1+BLUE*0.2);
13:     Point   V0=Point(500,500,500);                     //  視点の記述（座標：V0)
14:     Point   L0=Point(0,500,1000);                      //  照明光の記述（座標：L0)
15:     w.setview(V0);
16:
17:     //  球体（ポリゴン集合）の描画
18:     for(int i=0; i<f.n; i++){
19:         Polygon p=f.p[i];
20:         Vector  V=unit(p.p[0]-V0);                     //  視線の方向
21:         Vector  N=unit((p.p[0]-p.p[2])%(p.p[1]-p.p[3]));
22:         if( N*V <=0 ){
23:             Vector  L=unit(p.p[0]-L0);                 //  照明光の方向
24:             w.color( (max(-N*L,0)*0.8+0.2)*c );        //  描画色
25:             w.paint(p);                                //  ポリゴンの描画
26:         }
27:     }
28:     pause("EX5_3");
29: }
```

リスト5-4　EX5_4.cpp

```
 1: //  塗り重ね法を利用した陰線（陰面）処理
 2: //  トーラスを描画する
 3:
 4: #include        "graph3.h"
 5:
 6: //  aとbの値を入れ替える
 7: void    swap(int& a,int& b)      {  int     c=a;    a=b;    b=c;    }
 8: void    swap(float& a, float& b){  float   c=a;    a=b;    b=c;    }
 9:
10: main()
11: {
12:     Line    a= circle_xz( 18);                         //  トーラスの素（円）の生成
13:     Surface f= revolve_z( move(100,0,0)*50*a, 36 );    //  トーラス（ポリゴン集合）の生成
14:     Color   c= RED;
15:
16:     //  視点からの距離でソートする
17:     Point   V0=Point(500,500,500);                     //  視点の記述（座標：V0)
18:     float*  d= new float[f.n];
19:     int*    s= new int[f.n];
20:     for(int j=0; j<f.n; j++){
21:         Polygon p=f.p[j];
22:         d[j]=len(V0-(p.p[0]+p.p[1]+p.p[2]+p.p[3])/4);  //  視点とポリゴンの距離
23:         s[j]=j;                                        //  ポリゴン配列の添字
24:     }
25:
26:     //  視点とポリゴンの距離が大きい順に並び替える（バブルソート）
27:     //  距離が大きい順にポリゴン配列の添字がs[j]に格納される
28:     for(int k=0; k<f.n-1; k++)
29:         for( j=0; j<f.n-1; j++)
30:             if(d[j]<d[j+1])
31:                 swap(d[j],d[j+1]);
32:                 swap(s[j],s[j+1]);
33:             }
34:
35:     Win3D   w("EX5_4", WHITE*0.3+GREEN*0.1+BLUE*0.2);
36:     Point   L0=Point(0,500,1000);                      //  照明光の記述（座標：L0)
37:     w.setview(V0);
38:
39:     //  トーラスの描画
40:     for(int i=0; i<f.n; i++){
```

```
41:         Polygon p=f.p[s[i]];                              // 距離が大きい順に描画する
42:         Vector  L=unit(p.p[0]-L0);
43:         Vector  N=unit((p.p[0]-p.p[2])%(p.p[1]-p.p[3]));
44:         w.color( (max(-N*L,0)*0.8+0.2)*c );
45:         w.paint(p);
46:     }
47:     pause("EX5_4");
48: }
```

リスト5-5　EX5_5.cpp

```
 1: //  トーラスと球体を描画する(Ｚバッファ法)
 2:
 3: #include        "graph3.h"
 4:
 5: #define     X_SIZE      400                               //  ウィンドウサイズ
 6: #define     Y_SIZE      350
 7: float       z_buff[X_SIZE][Y_SIZE];                       //  Ｚバッファの定義
 8: #define     sup(p1,p2,a1,a2)   ((a1+a2<1)?p1:((a2)*p1+(a1)*p2)/(a1+a2)) //p1,p2 を補間する
 9:
10: //  ポリゴンを塗りつぶす
11: void    paint(Win3D& w, Polygon& a, Point& L0, Color& ca){
12:         Point   pp[10];
13:         for(int i=0; i<a.n; i++)pp[i]=w.screen(a.p[i]); //  screen 座標の計算
14:         Vector  V=unit(a.p[0]-w.v0);                      //  視線ベクトルの計算
15:         Vector  L=unit(a.p[0]-L0);                        //  照明ベクトルの計算
16:         Vector  N=unit((a.p[0]-a.p[2])%(a.p[1]-a.p[3]));//  法線ベクトルの計算
17:         w.color( (max(-N*L,0)*0.8+0.2)*ca );              //  描画色の設定
18:         int     y_max=pp[0].y, y_min=pp[0].y;             //  screen 座標の範囲(y_min,y_max)
19:         int     s0=0,   s1=0,   n=a.n-1;                  //  左端と右端の番号,頂点数
20:         for( i=0; i<a.n; i++){
21:             if(y_min>(int)pp[i].y)  {   y_min=pp[i].y;  s0=s1=i;    }
22:             else if(y_min==(int)pp[i].y)    s1=i;
23:             y_max=max(y_max, pp[i].y);
24:         }
25:         for(int y=y_min; y<=y_max; y++){
26:             if( pp[(s0-1+n)%n].y<y) s0=(s0-1+n)%n;
27:             if( pp[(s1+1)%n].y<y )  s1=(s1+1)%n;
28:             int     s2=(s1+1)%n,     s3=(s0-1+n)%n;
29:             Point   p0=pp[s0],  p1=pp[s1],  p2=pp[s2],  p3=pp[s3];
30:             Point   p4=sup(p1,p2,y-(int)p1.y,(int)p2.y-y);
31:             Point   p5=sup(p0,p3,y-(int)p0.y,(int)p3.y-y);
32:             for(int x=p4.x; x<=p5.x; x++){
33:                 Point   p=sup(p4,p5,x-p4.x,p5.x-x);     //  描画点座標の計算
34:                 if(z_buff[x][y]<p.z){ w.point(x,y); z_buff[x][y]=p.z; }
35:             }
36:         }
37: }
38:
39: main()
40: {
41:     for(int x=0; x<X_SIZE; x++)                           //  Ｚバッファの初期化
42:         for(int y=0; y<Y_SIZE; y++)z_buff[x][y]=-1e30;
43:     Win3D   w("EX5_5", X_SIZE, Y_SIZE, WHITE*0.3+GREEN*0.1+BLUE*0.2);
44:     w.setview(Point(500,500,500));                        //  視点の記述
45:     Point   L0=Point(0,500,1000);                         //  照明光の記述(座標：L0)
46:
47:     Line    a= circle_xz( 18);                            //  トーラスの素(円)の生成
48:     Surface f= revolve_z( move(100,0,0)*50*a, 36 );       //  トーラス(ポリゴン集合)の生成
49:     for(int i=0; i<f.n; i++)paint(w,f.p[i],L0,RED);       //  トーラスの描画
50:     a= circle_xz( 18, 0, 180);                            //  球体の素(半円)の生成
51:     f= move(0,100,0)*revolve_z( 70*a, 36 );               //  球体(ポリゴン集合)の生成
52:     for(i=0; i<f.n; i++)paint(w,f.p[i],L0,YELLOW);        //  球体の描画
53:
54:     pause("EX5_5");
55: }
```

# 6．シェーディングとポリゴンモデル

この章では，光学的モデルによる物体表面の明暗について考え，ポリゴンモデルに明暗を付けるシェーディング（shading）について考えていきます．次に，ポリゴンモデルを疑似的に曲面のように描画する方法であるグーローシェーディングおよびフォンシェーディング，さらに影付けの方法について考えていきます．

## 6.1　光学的モデル

３次元の形状モデルをより立体的に表現するためには，光源，視点，形状モデルの位置関係から表面の明るさを考える必要があります．そして，表面の明るさに応じた色でモデルを描画します．その表面の明るさの成分として，ディフューズ（拡散反射光:diffuse），スペキュラー（鏡面反射光:specular），アンビエント（環境光:ambient）があります．

（1）　ディフューズ（拡散反射光）

石膏像のように光沢のない反射が，ディフューズ（拡散反射光）です．図6-1に示すように，入射した光がすべての方向に同じ強さで反射します．その強さ（明るさ）は入射角によって決まり，式(6-1)で計算することができます．拡散反射による明るさ$r_d$は，光源の方向$L$，入射光の強さ$s$，拡散反射係数$k_d$によって決まります．すなわち，物体の表面に垂直な方向（$\alpha=0$）から光が入射した場合に明るく見え，垂直な方向から離れるにしたがって暗くなっていきます．逆のいい方をすると，法線ベクトル$N$は物体表面の方向を示しており，光源の方向に向いている（$\alpha$が小さい）場所は明るく見えます．そして，見る方向に対しては関係なく，同じ場所であればどの方向から見ても同じ明るさとなります．

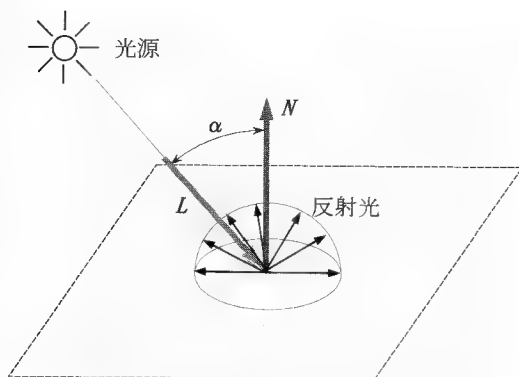



図6-1　ディフューズ

$$r_d = s \cdot k_d \cos\alpha \qquad (6\text{-}1)$$
$$\cos\alpha = -L \cdot N$$

$r_d$　:反射光の強さ

$s$　:入射光の強さ

$N$　:表面の法線ベクトル

$L$　:光源の方向ベクトル

$k_d$　:拡散反射係数

（2）　スペキュラー（鏡面反射光）

物体の表面は，拡散反射だけのものは少なく，光沢を持っているものが一般的です．光沢を表現するものが，図6-2に示すスペキュラー（鏡面反射光）です．鏡の場合，入射した光は$R$方向にのみ反射しますが，一般の物体の場合には多少乱反射があります．そのため，一方向（$R$方向）だけではなく，図6-2に示すように$R$方向を中心とした分布となるため，見る方向$V$によって明るさが変わります．$R$方向から見ると明るく見え，$R$方向から離れるにつれて暗く見えます．式(6-2)の$n$は反射光の分布を指定する係数で，明るく輝いて見えるハイライト効果の度合いを指定するものです．$n$が5, 10, 20, 40の場合の例を口絵に示します（金属などのスペキュラーを求める方法は，下式以外にも提案されています）．

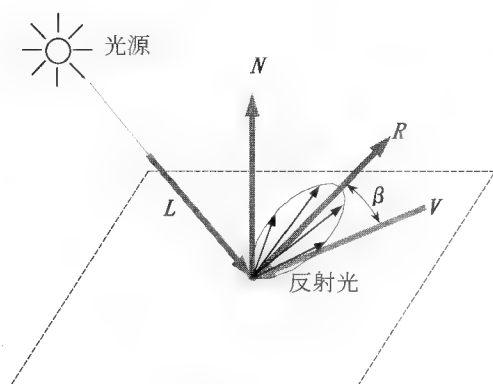



図6-2　スペキュラー

$$r_s = s \cdot k_s \cos^n \beta$$
$$\cos\beta = -R \cdot V \qquad (6\text{-}2)$$
$$R = L - 2N \cdot (L \cdot N)$$

$r_s$　:反射光の強さ
$s$　:入射光の強さ
$N$　:表面の法線ベクトル
$L$　:光源の方向ベクトル
$V$　:視点の方向ベクトル
$n$　:鏡面反射の強度係数
$k_s$　:鏡面反射係数

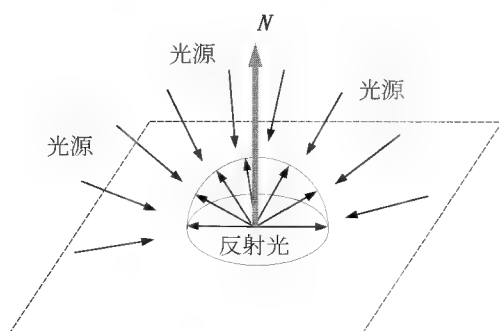



図6-3　アンビエント

$$r_e = k_e \qquad k_e\ :\text{環境反射係数} \qquad (6\text{-}3)$$

（3）　アンビエント（環境光）

一般に日中は，直接光が当たらなくても間接光が当たるので，物陰でも真っ暗で物が見えないところはほとんどありません．この場合，光源は1カ所ではなく，周囲にある物体からの反射光も光源となります．また，反射光が別の物体に当たり2次反射あるいは3次反射が起こるので，光源は複数あるものと考えなければなりません．このような光源に対する反射光を，図6-3に示すアンビエント（環境光）として表現します．すべての方向から均等な強さの光が入射するもの

とし，反射光もすべての方向に均等に発すると考えます．反射光の強さは，式(6-3)で表すものとします．

（4） ランバーモデル

明るさを計算する際に，ディフューズ（拡散反射光）とアンビエント（環境光）をについて考えているのがランバー（lambert）モデルです．次式で，描画に使用する色 $C$ を計算することができます．2章で説明したように，色の記述のためにColorクラスを用意しており，3原色の各成分の強さ（明るさ：0.0〜1.0）で記述しています．したがって，ベクトルと同様の方法（各成分ごとに計算する）で，描画に使用する色を計算することができます．

$$
\begin{aligned}
&C=(s\cdot k_d\cos\alpha+k_e)\cdot C_s\\
&\cos\alpha=-L\cdot N
\end{aligned}
\qquad (6\text{-}4)
$$

$C_s$ :物体の表面色

$C$ :描画に使用する色

（5） フォンモデル

明るさを計算する際に，ディフューズ（拡散反射光）とアンビエント（環境光）に加えて，スペキュラー（鏡面反射光）を考えているのがフォン（phong）モデルです．ディフューズとアンビエント成分については，ランバーモデルと同じですが，スペキュラーについては白（赤:1.0,緑:1.0,青:1.0）を用いることが多いようです．次式で，描画に使用する色 $C$ を計算することができます．

$$
\begin{aligned}
&C=(s\cdot k_d\cos\alpha+k_e)\cdot C_s+s\cdot k_s\cos^n\beta\cdot C_W\\
&\cos\alpha=-L\cdot N\\
&\cos\beta=2(L\cdot N)(N\cdot V)-L\cdot V
\end{aligned}
\qquad (6\text{-}5)
$$

$C_s$ :物体の表面色

$C_W$ :白色の成分（赤:1.0,緑:1.0,青:1.0）

$C$ :描画に使用する色

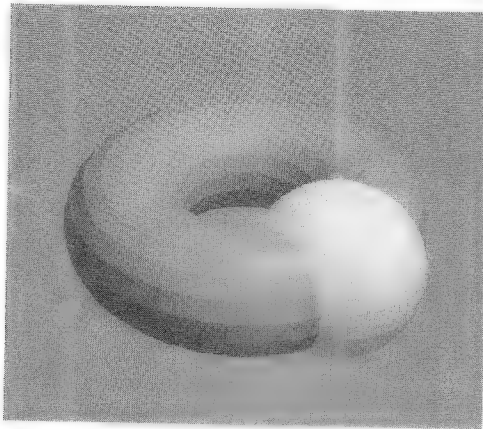



**図6-4** フラットシェーディング（EX6_1.cpp）

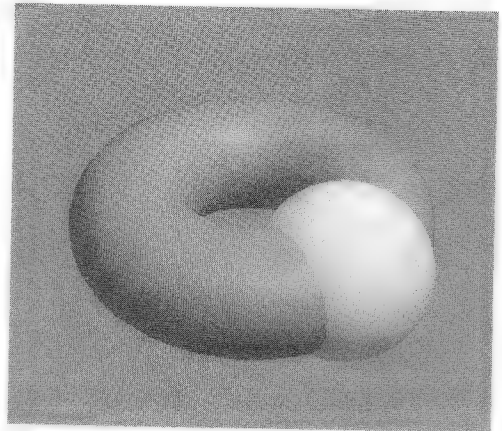



**図6-5** グーローシェーディング（EX6_2.cpp）

## 6.2　フラットシェーディング

ポリゴンモデルに明暗を付ける際に，モデルを構成するポリゴンごとに同色で塗りつぶす方法をフラットシェーディングまたはコンスタントシェーディングといいます．描画色を計算するためにはポリゴンの法線が必要になりますが，ポリゴンの頂点の座標から計算することができます．ランバーモデルを用いたフラットシェーディングは，すでに５章（EX5_3～5.cpp）で行っています．フォンモデルを用いてフラットシェーディングを行うプログラム例(EX6_1.cpp)をリスト6-1に，実行結果を図6-4に示します．shading関数で，式(6-5)の計算を行っています．

## 6.3　グーローシェーディング

ポリゴンモデルに明暗を付ける際に，フラットシェーディングではポリゴンごとに同色で塗りつぶしており，曲面を表現することは困難でした．もともとポリゴンモデルなので曲面ではありませんが，擬似的に曲面として表現する方法（スムーズシェーディング）が，グーローシェーディング（gouraud shading）やフォンシェーディング（phong shading）です．グーローシェーディングやフォンシェーディングを行うためには，ポリゴンデータ（頂点の位置）のほかに，ポリゴンの各頂点での法線ベクトル（ポリゴン平面の法線ベクトルではない）が必要になります．

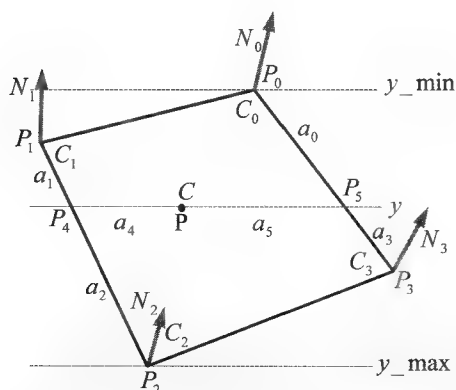
グーローシェーディングでは，はじめに図6-6に示すようにポリゴンの頂点の法線ベクトル（$N_0 \sim N_3$）から各頂点の描画色（$C_0 \sim C_3$）を計算します．そのあと，各頂点の描画色（$C_0 \sim C_3$）から線形補間することによって，ポリゴン内部の描画点Pの色$C$を求めます．

| | |
|---|---|
| $N_0 \sim N_3$ | ：各頂点での法線ベクトル |
| $P_0 \sim P_3$ | ：各頂点のスクリーン座標 |
| $C_0 \sim C_3$ | ：各頂点での描画色 |
| $C$ | ：P点での描画色 |
| P | ：描画を行う画素のスクリーン座標 |

**図6-6**　グーローシェーディングでの描画色

頂点$P_1, P_2$での描画色$C_1, C_2$から線形補間によって，色$C_4$を求めます．
同様に，頂点$P_0, P_3$での描画色$C_0, C_3$から線形補完によって，色$C_5$を求めます．

$$
\begin{aligned}
a_0 &= P_y - P_{0y} \\
a_1 &= P_y - P_{1y} \\
a_2 &= P_{2y} - P_y \\
a_3 &= P_{3y} - P_y
\end{aligned}
\qquad
\begin{aligned}
C_4 &= \frac{a_2 C_1 + a_1 C_2}{a_1 + a_2} \\
C_5 &= \frac{a_3 C_0 + a_0 C_3}{a_0 + a_3}
\end{aligned}
\tag{6-6}
$$

ここ，色$C_4, C_5$から線形補間によって，目的とする描画点の描画色$C$を求めます．

$$a_4 = P_x - P_{4x} \qquad P_4 = \frac{a_2 P_1 + a_1 P_2}{a_1 + a_2}$$
$$a_5 = P_{5x} - P_x \qquad P_5 = \frac{a_3 P_0 + a_0 P_3}{a_0 + a_3} \tag{6-7}$$

$$C = \frac{a_5 C_4 + a_4 C_5}{a_4 + a_5} \tag{6-8}$$

グーローシェーディングを適用したプログラム例(EX6_2.cpp)をリスト6-2に，実行結果を図6-6に示します．プログラムでは，paint関数でポリゴンを描画しています．paint関数では，図6-5に示すようにポリゴン内部の点$\mathrm{P} = (x, y)$を，以下に示す方法で描画していきます．

① ポリゴンの各頂点を，スクリーン座標($P_0 \sim P_3$)に変換する(screen関数を使用)．
　各頂点での描画色($C_0 \sim C_3$)を計算する(shading関数を使用)．

② 変換後の各頂点の$Y$座標（$P_{0y} \sim P_{3y}$）の範囲（$y_\mathrm{min} \sim y_\mathrm{max}$）を求める．

③ 各$Y$座標（$y_\mathrm{min} \sim y_\mathrm{max}$）での範囲$P_4$(左端)$\sim P_5$(右端)，$C_4$および$C_5$を求める．
　$P_4, P_5, C_4, C_5$は上式で計算する(sup関数を使用)．

④ $P_4 \sim P_5$（$P_{4x} \sim P_{5x,y}$）までの各点$\mathrm{P} = (x, y)$の描画色($C$)を求める．
　$C$は上式で計算する(sup関数を使用)．

⑤ 描画色($C$)を描画する．

## 6.4　フォンシェーディング

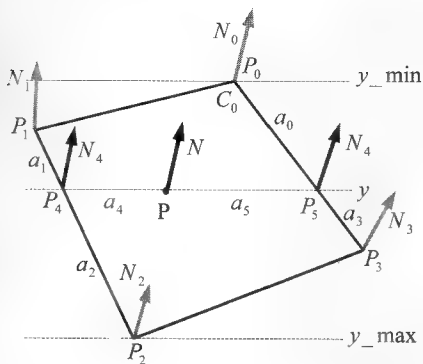
グーローシェーディングでは，描画点の色をポリゴンの各頂点の色から線形補間することによって求めていました．それに対してフォンシェーディングでは，図6-7に示すように描画色を計算する際に使用する法線ベクトル$N$をポリゴンの各頂点の法線ベクトル（$N_0 \sim N_3$）から線形補間することによって，次式を用いて求めます．

$N_0 \sim N_3$　：各頂点での法線ベクトル
$P_0 \sim P_3$　：各頂点のスクリーン座標
$N$　：P点での法線ベクトル
P　：描画を行う画素のスクリーン座標

図6-7　フォンシェーディングでの描画色

法線ベクトル$N_1, N_2$から線形補間によって，ベクトル$N_4$を求めます．
同様に，法線ベクトル$N_0, N_3$から線形補間によって，ベクトル$N_5$を求めます．
$a_0 \sim a_3$は，グーローシェーディングの場合と同様です．

$$N_4 = \frac{a_2 N_1 + a_1 N_2}{a_1 + a_2} \qquad N_5 = \frac{a_3 N_0 + a_0 N_3}{a_0 + a_3} \tag{6-9}$$

次に，ベクトル$N_4, N_5$から線形補間によって，法線ベクトル$N$を求めます．
求めた法線ベクトル$N$を使って，描画点の描画色を計算します．
$a_4 \sim a_5$は，グーローシェーディングの場合と同様です．

$$N = \frac{a_5 N_4 + a_4 N_5}{\left| a_5 N_4 + a_4 N_5 \right|} \tag{6-10}$$

フォンシェーディングを適用したプログラムをリストリスト6-3に，実行結果(EX6_3.cpp)を図6-8に示します．

プログラムでは，paint関数でポリゴンを描画しています．paint関数では，図6-7に示すようにポリゴン内部の点$\mathrm{P} = (x, y)$を，以下に示す方法で描画していきます．

① ポリゴンの各頂点を，スクリーン座標($P_0 \sim P_3$)に変換する(screen関数を使用)．
② 変換後の各頂点のY座標($P_{0_y} \sim P_{3_y}$)の範囲（$y$_min～$y$_max）を求める．
③ 各Y座標（$y$_min～$y$_max）での範囲$P_4$(左端)～$P_5$(右端)，$N_4$および$N_5$を求める．
　$P_4, P_5, N_4, N_5$は上式で計算する(sup関数を使用)．
④ $P_4 \sim P_5$ ($P_{4_x} \sim P_{5_{x,y}}$)までの各点$P = (x, y)$の法線ベクトル($N$)を求める．
　$N$は上式で計算する(sup関数を使用)．
⑤ 法線ベクトル($N$)を使用して，描画色を求めて描画する．

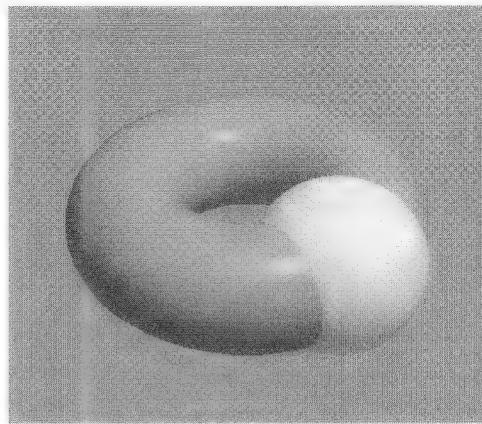



**図6-8**　フォンシェーディング（EX6_3.cpp）

フラットシェーデング,グーローシェーデング,フォンシェーデングにおける法線ベクトルの使用を図示すると，下図のようになります(灰色矢印の法線ベクトルは，描画時に補間して求める).

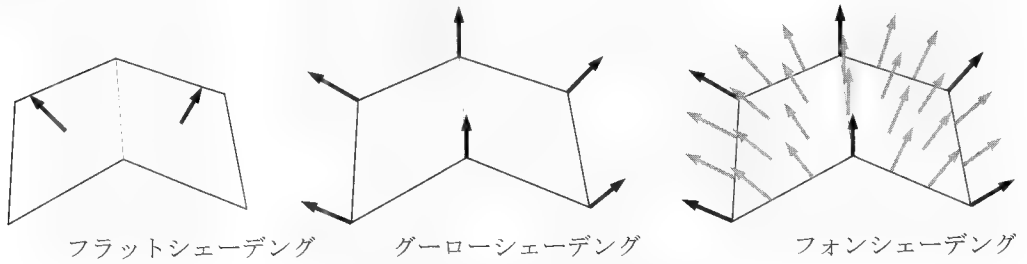



**図6-9**　法線ベクトルの使用方法

## 6.5　影付け

ポリゴンモデルの場合の影付け(shadowing)の方法について，考えてみましょう．光源からの照明光が当たる場合には，描画色を計算するために前述シェーディングを行います．一方，ほかのポリゴンにさえぎられて照明光が当たらない場合（影になる場合）には，ディフューズ（拡散反射光）とスペキュラー（鏡面反射光）はなく，アンビエント（環境光）のみと考えます（$s=0$）．描画対象としているポリゴン内の描画点に，光源からの照明光が当たるか否かを判定する方法が必要になります.

（1）　シャドウマップ法

シャドウマップ法では,光源から照明光がとどく範囲をシャドウバッファに記録します．$Z$バッファ法と同様に，図のように光源を視点として，スクリーン座標の$Z$座標（あるいは光源からの距離）をシャドウバッファに記録します（記録前にシャドウバッファは，光源から十分遠方となる値で初期化しておきます）．描画対象としているポリゴン内の描画点に対し，光源から見た$Z$座標（あるいは光源からの距離）を求めます．描画点に対応するシャドウバッファの値よりも，その値が光源から遠い値の場合に，影とします．図6-10に示すように，描画点Pに対応するシャドウバッファに，物体の点Qの値が記録されている場合には影となります.

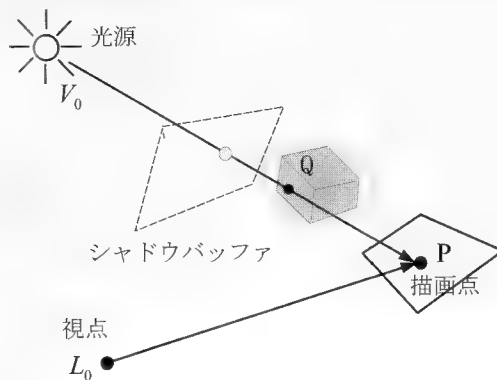



**図6-10**　シャドウマップ法

（２）　シャドウボリューム法

図6-11に示すように，物体によってできる影の空間を，シャドウボリューム(shadow volume)と呼びます．描画対象としているポリゴン内の描画点が，シャドウボリューム内にある場合には影ということになります．あるいは，シャドウボリュームを構成するポリゴンと視線（視点と描画点を結ぶ線）との交差によって，判断することができます．

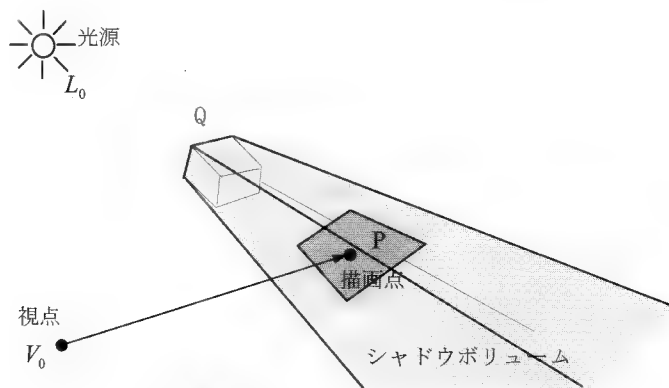



図6-11　シャドウボリューム法

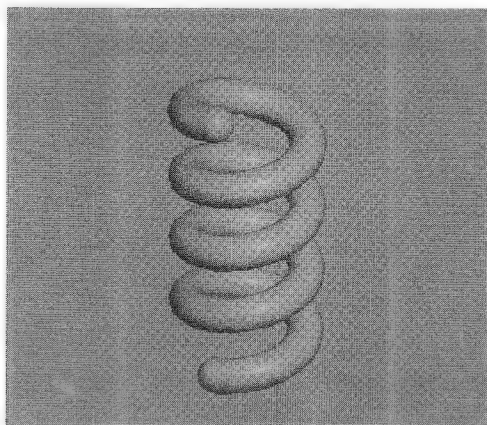
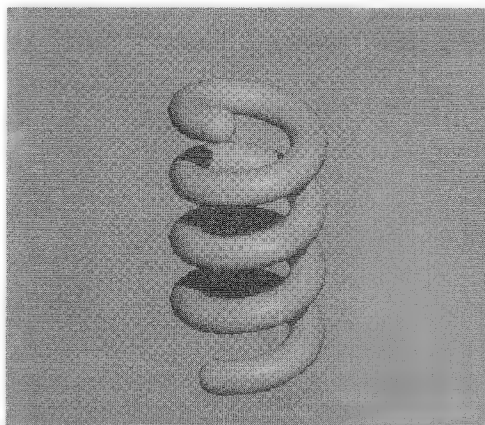
図6-2にシャドウマップ法により影付けをしたスプリングの描画例(AP2_1.cpp)を示します．（付録4-2参照）影付けなしの画像および影付け画像を表示します．また，図6-13にコーヒーカップの描画例(AP2_2.cpp)を示します．ワイヤーフレームによる描画，フラットシェーディング，グーロシェーディング，フォンシェーディングにシャドウマップ法により影付けをした描画です．

①影なし

②影付け

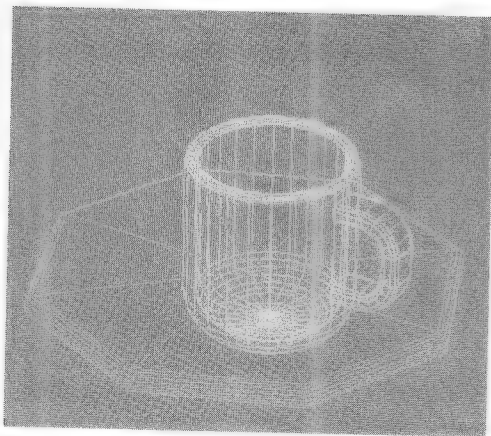
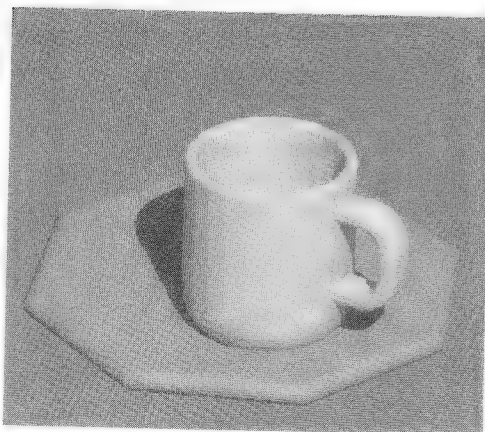
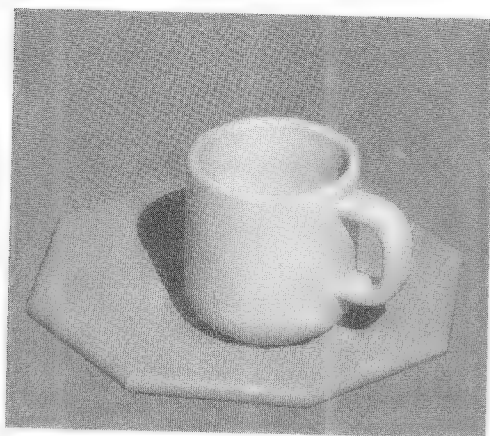
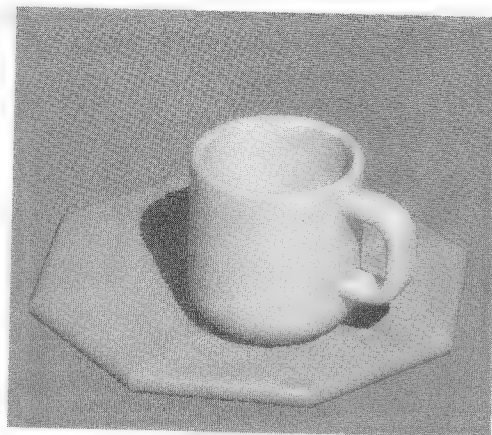
図6-12　スプリング（AP2_1.cpp）

①ワイヤーフレーム

②フラットシェーディング

③グーローシェーディング

④フォンシェーディング

**図6-13**　コーヒーカップ(AP2_2.cpp)

## 演習問題

**6-1**　プログラム例EX6_1.cpp,EX6_2.cpp,EX6_3.cppを変更して，ポリゴン数を少なくした場合について違いを観察せよ(ポリゴン数は，main内のcircle_xz,rezolve_z関数の引数で変わります)．

**6-2**　プログラム例EX6_3.cppを変更して，式(6-5)中のスペキュラー（鏡面反射光）成分の項 $k_s \cos^n \beta \cdot C_W$ の $n$ を変更(5, 10, 20, 40)した場合の違いを観察せよ．また，スペキュラー成分のないランバーモデルについても観察せよ．

（$n$=5とする場合には，shading 関数内を以下のように変更する）

```
return  kd*max(-N*L,0)*c+ks*pow(max(-R*V,0),5)*WHITE+ke*c;
```

リスト 6-1　EX6_1.cpp

```
 1: //  トーラスと球を描画する(フラットシェーディング)
 2: #include    "graph3.h"
 3:
 4: #define     X_SIZE  400                                  //  ウィンドウサイズ
 5: #define     Y_SIZE  350
 6: float       z_buff[X_SIZE][Y_SIZE];                      //  Zバッファの定義
 7: #define     sup(p1,p2,a1,a2)     ((a1+a2<1)?p1:((a2)*p1+(a1)*p2)/(a1+a2)) //p1,p2を補間する
 8:
 9: //  反射光を計算する    V:視線ベクトル  L:照明光の方向ベクトル  N:表面の法線ベクトル
10: Color   shading(Vector& V, Vector& L, Vector& N, Color& c){
11:         float   kd=0.7, ks=0.7, ke=0.3;                  //拡散反射係数,鏡面反射係数,環境光
12:         Vector  R=L-2*(L*N)*N;                           //  反射方向ベクトル
13:         return  kd*max(-N*L,0)*c+ks*pow(max(-R*V,0),20)*WHITE+ke*c;
14: }
15:
16: //  ポリゴンを塗りつぶす
17: void    paint(Win3D& w, Polygon& a, Point& L0, Color& ca){
18:         Point   pp[10];
19:         for(int i=0; i<a.n; i++)pp[i]=w.screen(a.p[i]); //  screen座標の計算
20:         Vector  V=unit(a.p[0]-w.v0);                     //  視線ベクトルの計算
21:         Vector  L=unit(a.p[0]-L0);                       //  照明ベクトルの計算
22:         Vector  N=unit((a.p[0]-a.p[2])%(a.p[1]-a.p[3]));//  法線ベクトルの計算
23:         w.color( shading( V,L,N,ca ) );                  //  描画色の設定
24:         int     y_max=pp[0].y, y_min=pp[0].y;            //  screen座標の範囲を求める
25:         int     s0=0,    s1=0,    n=a.n-1;               //  左端と右端の番号,頂点数
26:         for( i=0; i<a.n; i++){
27:             if(y_min>(int)pp[i].y)  {   y_min=pp[i].y;  s0=s1=i;    }
28:             else if(y_min==(int)pp[i].y)    s1=i;
29:             y_max=max(y_max, pp[i].y);
30:         }
31:         for(int y=y_min; y<=y_max; y++){
32:             if( pp[(s0-1+n)%n].y<y) s0=(s0-1+n)%n;
33:             if( pp[(s1+1)%n].y<y )  s1=(s1+1)%n;
34:             int     s2=(s1+1)%n,    s3=(s0-1+n)%n;
35:             Point   p0=pp[s0],  p1=pp[s1],  p2=pp[s2],  p3=pp[s3];
36:             Point   p4=sup(p1,p2,y-(int)p1.y,(int)p2.y-y);
37:             Point   p5=sup(p0,p3,y-(int)p0.y,(int)p3.y-y);
38:             for(int x=p4.x; x<=p5.x; x++){
39:                 Point   p=sup(p4,p5,x-p4.x,p5.x-x);      //  描画点座標の計算
40:                 if(z_buff[x][y]<p.z){ w.point(x,y); z_buff[x][y]=p.z; }
41:             }
42:         }
43: }
44:
45: main()
46: {
47:     for(int x=0; x<X_SIZE; x++)                          //  Zバッファの初期化
48:         for(int y=0; y<Y_SIZE; y++)z_buff[x][y]=-1e30;
49:     Win3D   w("EX6_1", X_SIZE, Y_SIZE, WHITE*0.3+GREEN*0.1+BLUE*0.2);
50:     w.setview(Point(500,500,500));                       //  視点の記述
51:     Point   L0=Point(0,500,1000);                        //  照明光の記述（座標：L0)
52:
53:     Line    a= circle_xz( 18);                           //  トーラスの素（円）の生成
54:     Surface f= revolve_z( move(100,0,0)*50*a, 36 );      //  トーラス（ポリゴン集合）の生成
55:     for(int i=0; i<f.n; i++)paint(w,f.p[i],L0,RED);      //  トーラスの描画
56:
57:     a= circle_xz( 18, 0, 180);                           //  球体の素（半円）の生成
58:     f= move(0,100,0)*revolve_z( 70*a, 36 );              //  球体（ポリゴン集合）の生成
59:     for(i=0; i<f.n; i++)paint(w,f.p[i],L0,YELLOW);       //  球体の描画
60:     pause("EX6_1");
61: }
```

リスト 6-2　EX6_2.cpp

```
 1: //  トーラスと球を描画する(グーローシェーディング)
 2:
 3: #include        "graph3.h"
 4:
 5: #define     X_SIZE  400                                  //  ウィンドウサイズ
 6: #define     Y_SIZE  350
 7: float       z_buff[X_SIZE][Y_SIZE];                      //  Zバッファの定義
 8: #define     sup(p1,p2,a1,a2)     ((a1+a2<1)?p1:((a2)*p1+(a1)*p2)/(a1+a2)) //p1,p2 を補間する
 9:
10: //  反射光を計算する    V:視線ベクトル  L:照明光の方向ベクトル  N:表面の法線ベクトル
11: Color   shading(Vector& V, Vector& L, Vector& N, Color& c){
12:         float   kd=0.7, ks=0.7, ke=0.3;                  //拡散反射係数,鏡面反射係数,環境光
13:         Vector  R=L-2*(L*N)*N;                           //  反射方向ベクトル
14:         return  kd*max(-N*L,0)*c+ks*pow(max(-R*V,0),20)*WHITE+ke*c;
15: }
16:
17: //  ポリゴンを塗りつぶす
18: void    paint(Win3D& w, Polygon& a, Polygon& an, Point& L0, Color& ca){
19:         Point   pp[10];
20:         Color   cc[10];
21:         for(int i=0; i<a.n; i++){
22:             pp[i]=w.screen(a.p[i]);                      //  screen 座標の計算
23:             Vector  V=unit(a.p[i]-w.v0);                 //  視線ベクトルの計算
24:             Vector  L=unit(a.p[i]-L0);                   //  照明ベクトルの計算
25:             cc[i]=shading( V,L,an.p[i],ca );
26:         }
27:         int     y_max=pp[0].y, y_min=pp[0].y;            //  screen 座標の範囲を求める
28:         int     s0=0,   s1=0,   n=a.n-1;                 //  左端と右端の番号,頂点数
29:         for( i=0; i<a.n; i++){
30:             if(y_min>(int)pp[i].y)   {   y_min=pp[i].y;  s0=s1=i;    }
31:             else if(y_min==(int)pp[i].y)    s1=i;
32:             y_max=max(y_max, pp[i].y);
33:         }
34:         for(int y=y_min; y<=y_max; y++){
35:             if( pp[(s0-1+n)%n].y<y) s0=(s0-1+n)%n;
36:             if( pp[(s1+1)%n].y<y )  s1=(s1+1)%n;
37:             int     s2=(s1+1)%n,    s3=(s0-1+n)%n;
38:             Point   p0=pp[s0],  p1=pp[s1],  p2=pp[s2],  p3=pp[s3];
39:             Color   c0=cc[s0],  c1=cc[s1],  c2=cc[s2],  c3=cc[s3];
40:             Point   p4=sup(p1,p2,y-(int)p1.y,(int)p2.y-y);
41:             Color   c4=sup(c1,c2,y-(int)p1.y,(int)p2.y-y);
42:             Point   p5=sup(p0,p3,y-(int)p0.y,(int)p3.y-y);
43:             Color   c5=sup(c0,c3,y-(int)p0.y,(int)p3.y-y);
44:             for(int x=p4.x; x<=p5.x; x++){
45:                 Point   p=sup(p4,p5,x-p4.x,p5.x-x);      //  描画点座標の計算
46:                 w.color(  sup(c4,c5,x-p4.x,p5.x-x));
47:                 if(z_buff[x][y]<p.z){ w.point(x,y); z_buff[x][y]=p.z; }
48:             }
49:         }
50: }
51:
52: main()
53: {
54:     for(int x=0; x<X_SIZE; x++)                          //  Zバッファの初期化
55:         for(int y=0; y<Y_SIZE; y++)z_buff[x][y]=-1e30;
56:     Win3D   w("EX6_2", X_SIZE, Y_SIZE, WHITE*0.3+GREEN*0.1+BLUE*0.2);
57:     w.setview(Point(500,500,500));                       //  視点の記述
58:     Point   L0=Point(0,500,1000);                        //  照明光の記述 (座標 : L0)
59:
60:     Line    a= circle_xz( 18);                           //  トーラスの素 (円) の生成
61:     Surface f= revolve_z( move(100,0,0)*50*a, 36 );      //  トーラス (ポリゴン集合) の生成
62:     Surface fn=revolve_z( a, 36 );                       //  トーラスの法線の生成
63:     for(int i=0; i<f.n; i++)paint(w,f.p[i],fn.p[i],L0,RED);  //  トーラスの描画
64:
65:     a= circle_xz( 18, 0, 180);                           //  球体の素 (半円) の生成
66:     f= move(0,100,0)*revolve_z( 70*a, 36 );              //  球体 (ポリゴン集合) の生成
67:     fn=revolve_z( a, 36 );                               //  球体の法線の生成
```

```
68:     for(i=0; i<f.n; i++)paint(w,f.p[i],fn.p[i],L0,YELLOW);  //  球体の描画
69:     pause("EX6_2");
70: }
```

リスト 6-3　EX6_3.cpp

```
 1: //  トーラスと球を描画する(フォンシェーディング)
 2:
 3: #include        "graph3.h"
 4:
 5: #define     X_SIZE  400                              //  ウィンドウサイズ
 6: #define     Y_SIZE  350
 7: float       z_buff[X_SIZE][Y_SIZE];                  //  Zバッファの定義
 8: #define     sup(p1,p2,a1,a2)    ((a1+a2<1)?p1:((a2)*p1+(a1)*p2)/(a1+a2)) //p1,p2 を補間する
 9:
10: //  反射光を計算する    V:視線ベクトル  L:照明光の方向ベクトル  N:表面の法線ベクトル
11: Color   shading(Vector& V, Vector& L, Vector& N, Color& c){
12:         float   kd=0.7, ks=0.7, ke=0.3;              //拡散反射係数,鏡面反射係数,環境光
13:         Vector  R=L-2*(L*N)*N;                       //  反射方向ベクトル
14:         return  kd*max(-N*L,0)*c+ks*pow(max(-R*V,0),20)*WHITE+ke*c;
15: }
16:
17: //  ポリゴンを塗りつぶす
18: void    paint(Win3D& w, Polygon& a, Polygon& na, Point& L0, Color& ca){
19:         Point   pp[10];
20:         for(int i=0; i<a.n; i++)pp[i]=w.screen(a.p[i]); //  screen 座標の計算
21:         int     y_max=pp[0].y, y_min=pp[0].y;          //  screen 座標の範囲を求める
22:         int     s0=0,   s1=0,   n=a.n-1;               //  左端と右端の番号,頂点数
23:         for( i=0; i<a.n; i++){
24:             if(y_min>(int)pp[i].y)  {   y_min=pp[i].y;  s0=s1=i;    }
25:             else if(y_min==(int)pp[i].y)    s1=i;
26:             y_max=max(y_max, pp[i].y);
27:         }
28:         for(int y=y_min; y<=y_max; y++){
29:             if( pp[(s0-1+n)%n].y<y) s0=(s0-1+n)%n;
30:             if( pp[(s1+1)%n].y<y )  s1=(s1+1)%n;
31:             int     s2=(s1+1)%n,    s3=(s0-1+n)%n;
32:             Point   p0=pp[s0],  p1=pp[s1],  p2=pp[s2],  p3=pp[s3];
33:             Vector  n0=na.p[s0],n1=na.p[s1],n2=na.p[s2],n3=na.p[s3];
34:             Point   p4=sup(p1,p2,y-(int)p1.y,(int)p2.y-y);
35:             Vector  n4=sup(n1,n2,y-(int)p1.y,(int)p2.y-y);
36:             Point   p5=sup(p0,p3,y-(int)p0.y,(int)p3.y-y);
37:             Vector  n5=sup(n0,n3,y-(int)p0.y,(int)p3.y-y);
38:             for(int x=p4.x; x<=p5.x; x++){
39:                 Point   p=sup(p4,p5,x-p4.x,p5.x-x);      //  描画点座標の計算
40:                 Vector  N=unit(sup(n4,n5,x-p4.x,p5.x-x));// 法線ベクトルの計算
41:                 Vector  V=unit(w.world(p)-w.v0);         //  視線ベクトルの計算
42:                 Vector  L=unit(w.world(p)-L0);           //  照明ベクトルの計算
43:                 w.color(shading( V,L,N,ca ));
44:                 if(z_buff[x][y]<p.z){ w.point(x,y); z_buff[x][y]=p.z; }
45:             }
46:         }
47: }
48:
49: main()
50: {
51:     for(int x=0; x<X_SIZE; x++)                          //  Zバッファの初期化
52:         for(int y=0; y<Y_SIZE; y++)z_buff[x][y]=-1e30;
53:     Win3D   w("EX6_3", X_SIZE, Y_SIZE, WHITE*0.3+GREEN*0.1+BLUE*0.2);
54:     w.setview(Point(500,500,500));                       //  視点の記述
55:     Point   L0=Point(0,500,1000);                        //  照明光の記述 (座標 : L0)
56:
57:     Line    a= circle_xz( 18);                           //  トーラスの素 (円) の生成
58:     Surface f= revolve_z( move(100,0,0)*50*a, 36 );      //  トーラス (ポリゴン集合) の生成
59:     Surface fn=revolve_z( a, 36 );                       //  トーラスの法線の生成
60:     for(int i=0; i<f.n; i++)paint(w,f.p[i],fn.p[i],L0,RED); //  トーラスの描画
61:
```

```
a= circle_xz( 18, 0, 180);                          // 球体の素（半円）の生成
f= move(0,100,0)*revolve_z( 70*a, 36 );              // 球体（ポリゴン集合）の生成
fn=revolve_z( a, 36 );                               // 球体の法線の生成
for(i=0; i<f.n; i++)paint(w,f.p[i],fn.p[i],LO,YELLOW);  // 球体の描画
pause("EX6_3");
```

# 7．レイトレーシング

いままでは，物体の形状を表すためにポリゴンで構成したポリゴンモデルを用いてきましたが，この章ではソリッドモデルを使用します．物体の形状を数学的関数で表し，レイトレーシング（光線追跡法:ray tracing algorithm）を用いて描画します．

光源からの光（照明光）が物体に当たり，反射することによって，その物体は見えます．下図のように，スクリーン（画面）上の画素（色）は，物体からの反射光によるものと考えることができます．レイトレーシングでは，スクリーンを構成する画素ごとに物体からの反射光を，下図とは逆方向(視点から物体方向)に追跡していきます．

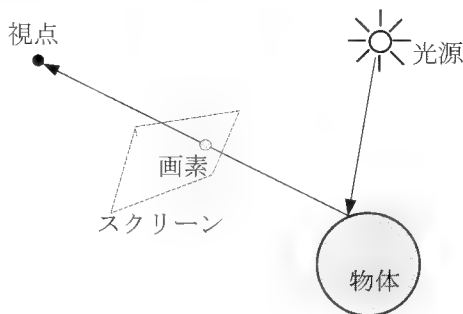



図7-1　レイトレーシング

## 7.1　球体の描画

下図に示すように，球体（中心点座標 $B_0$，半径 $r$）の描画について考えます．まず，視点 $V_0$ とスクリーン上の画素(描画点)との位置関係から，視線ベクトル $V$ を求めます．そして，視線が球体と交わる場合，光源との位置関係から交点Pの色を求めて描画します．

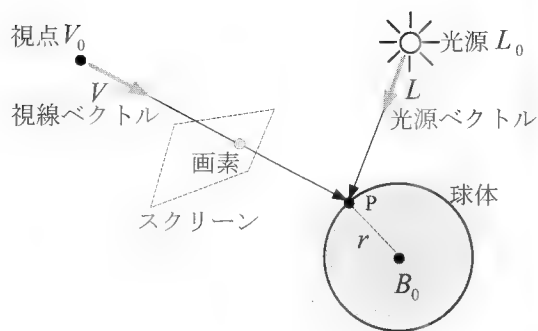



図7-2　球体の描画

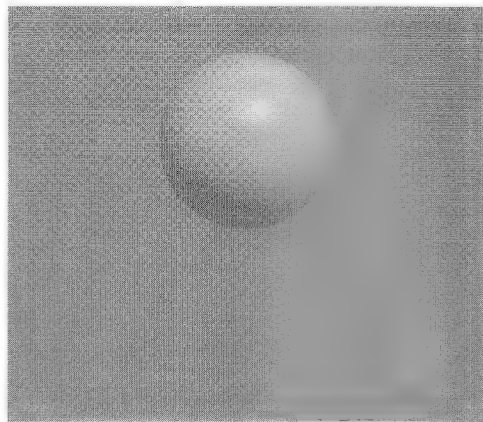



図7-3　球体の描画(EX7_1.cpp)

視線が球体と交わるか判定する方法について考えましょう．視線ベクトル$V$と交点Pとの関係，および半径$r$の関係から次式が成り立ちます（$t$は視点から交点までの距離を意味する．P,$V_0$,$B_0$は各点の座標を表すとともに，原点から各点までのベクトルを表すものとする）．

$$
\begin{aligned}
&V_0 + t \cdot V = P \\
&(P - B_0) \cdot (P - B_0) = r^2
\end{aligned}
\tag{7-1}
$$

上式から次の2次式が得られ，その判別式$d$は次式になります．

$$t^2 + 2t \cdot V \cdot (V_0 - B_0) + (V_0 - B_0) \cdot (V_0 - B_0) - r^2 = 0 \tag{7-2}$$

$$d = \{V \cdot (V_0 - B_0)\}^2 - (V_0 - B_0) \cdot (V_0 - B_0) + r^2 \tag{7-3}$$

判別式$d$が0以上の場合には球体と交わり，そのときの交点（視点に近い交点）の座標は下式で求められます．

$$
\begin{aligned}
&t = -V \cdot (V_0 - B_0) - \sqrt{d} \\
&P = V_0 + t \cdot V
\end{aligned}
\tag{7-4}
$$

視線が球体と交わる場合，交点Pと光源$L_0$との位置関係から光源ベクトル$L$を求め，また球体の中心点$B_0$と交点Pとの位置関係から法線ベクトル$N$を求めます．視線ベクトル$V$,光源ベクトル$L$,法線ベクトル$N$から，6章で説明したフォンモデルなどにより物体の色(描画色)を計算することができます（$L_0$は座標を表すとともに，原点からのベクトルを表すものとする）．

$$L = \frac{P - L_0}{|P - L_0|} \tag{7-5}$$

$$N = \frac{P - B_0}{|P - B_0|} \tag{7-6}$$

レイトレーシング法を用いて球体を描画するプログラム例(EX7_1.cpp)をリスト7-1に，実行結果を図7-3に示します．プログラム例では，光線を記述するためのRayクラス（始点と方向）と，球体を記述するためのBallクラス（中心点と半径）を用意して使用しています．描画を行う画素のスクリーン座標（プログラム中ではmain関数の$x$, $y$）を，world関数によって標準座標に変換して視線ベクトル$V$を計算します．次に，Ballクラスのhit関数によって視線$VR$が球体$B_a$と交わるか判定します．視線が球体と交わる場合には，shading関数によって球体の色を計算して描画します．

## 7.2 平面（ポリゴン）の描画

ポリゴン(平面上の多角形)の描画について考えましょう．図7-4に示すように，視線がポリゴン（平面上の点 $P_0$，法線ベクトル $N$）と交わる場合，光源 $L_0$ との位置関係からポリゴンの色を求めて描画します．

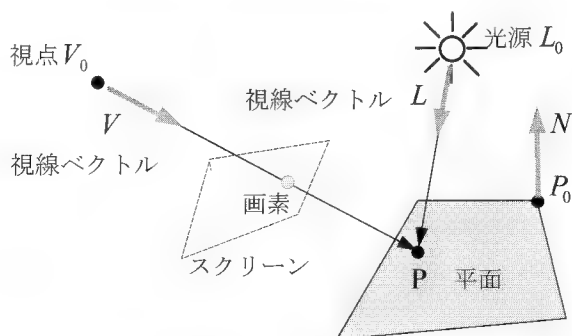



図7-4　平面の描画

まず，視線とポリゴンを含む平面との交点Pを求めます．視線ベクトル $V$ と交点Pとの関係，および平面上の点 $P_0$ の関係から次式が成り立ちます（$t$ は視点から交点までの距離を意味する．P, $P_0$ は各点の座標を表すとともに，原点から各点までのベクトルを表すものとする）．

$$\begin{aligned} &V_0 + t \cdot V = P \\ &(P_0 - P) \cdot N = 0 \end{aligned} \tag{7-7}$$

上式から交点Pを求めることができます．

$$\begin{aligned} &t = \frac{(P_0 - V_0) \cdot N}{V \cdot N} \\ &P = V_0 + t \cdot V \end{aligned} \tag{7-8}$$

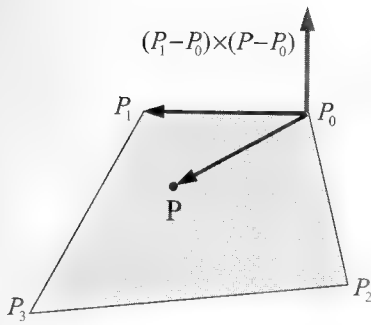
次に，求めた交点がポリゴン内部にあるかどうかを，次式の条件を満たすか否かによって判定します．図7-5のようにポリゴンの各辺をベクトルとした場合，交点Pが各辺の左側に位置することを判定しています．この判定方法は，凸ポリゴンの場合に使用できます．

$$
\begin{aligned}
\{(P_1-P_0)\times(P-P_0)\}\cdot N \geq 0\\
\{(P_2-P_1)\times(P-P_1)\}\cdot N \geq 0\\
\{(P_3-P_2)\times(P-P_2)\}\cdot N \geq 0\\
\{(P_0-P_3)\times(P-P_3)\}\cdot N \geq 0
\end{aligned}
\qquad (7\text{-}9)
$$

図7-5　交点の判定

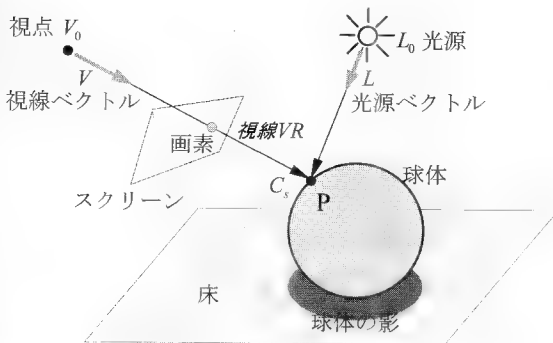



図7-6　球体と床の描画

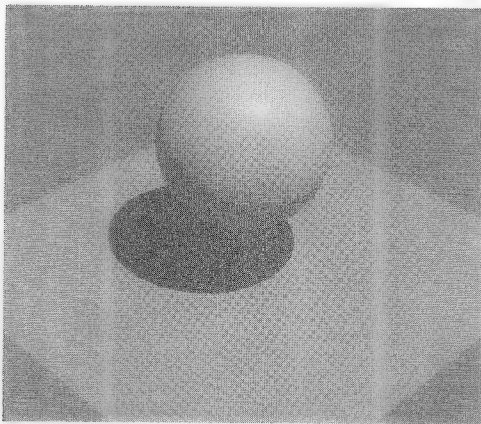



図7-7　球体と床の描画(EX7_2.cpp)

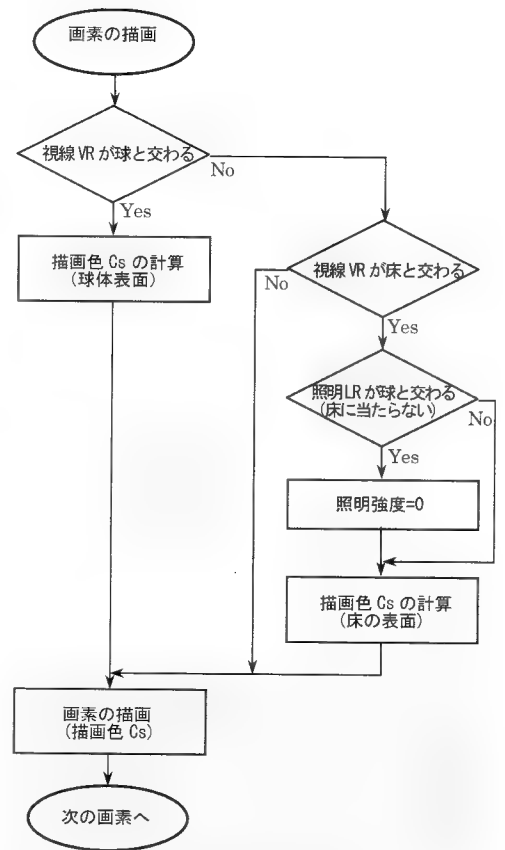



図7-8　球体と床の描画

球体と床を描画するプログラム例(EX7_2.cpp)をリスト7-2に，実行結果を図7-7に示します．

床（壁）を記述するために，Wallクラス（ポリゴン）を用意して使用しています．Ballクラスと同様に，hit関数によって視線と交わるか判定します．

スクリーンを構成する画素の描画手順を，図7-8のフローチャートに示します．物体が複数になると，ほかの物体の日陰（照明光がほかの物体にさえぎられて，当たらない）になる場合があります．そのため，視線$VR$から見えるところ(視線との交点P)はどこで，またそこには照明光が当たっているかどうかを考える必要があります．そこで，照明光の強度を変数$s$（初期値1）で記述しており，交点Pの色$C_s$の計算に使用しています．一般的に物体が複数ある場合には，視線$VR$と各物体との交点のうち視点$V_0$に最も近いものを選び出して描画する必要があります．

リスト7-2のプログラム例(EX7_2.cpp)で記述されているRayクラス，Ballクラス，Wallクラス，shading関数を，ヘッダファイルray.h(巻末に示す)として用意しておきます．

## 7.3　鏡面反射光の表現

表面がつるつるしていて，周囲の物体が写り込むような物体の描画について考えてみましょう．下図に示すように，視線の反射方向$R$の物体が写り込んでいるものと考えることができます．したがって，視点を交点Pとして$R$方向を見る必要があり，交点Pの色$C_s$（球体の色）に

図7-9　球体と床の描画

図7-10 反射する物体(EX7_3.cpp)

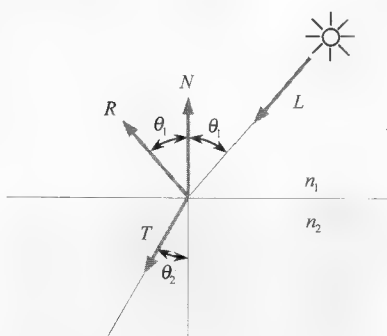
図7-11　球と床の描画

交点Pから反射方向 $R$ に見える色 $C_r$（床の色）に反射率 $k_r$ を考慮して加える操作をします．また，交点Pから反射方向 $R$ に見える物体にも写り込みがある場合には，その物体でも同様の操作をする必要があります．視線の反射方向 $R$ は次式で求めることができます．

$$R = 2N \cdot (V \cdot N) - V$$
$$C = C_s + k_r \cdot C_r \qquad (7\text{-}10)$$

球体と床を描画するプログラム例(EX7_3.cpp)をリスト7-3に，実行結果を図7-10に示します．球体に床が写っているのがわかります．スクリーンを構成する画素の描画手順を，図7-11のフローチャートに示します．視線が球体と交わる場合，反射視線の $R$ 方向の色を考慮しているところが追加されています．

## 7.4　透過光の表現

ガラス玉やプリズムのように，光が透過する物体について考えてみましょう．光が屈折率の異なる物質に入る場合，図7-12に示すように反射および屈折（光線の方向が変わる）が起こります．屈折率は，物質によって固有の値(空気:1.00，水:1.33，ガラス:1.5～1.7)を持っています．そして，屈折率 $n_1$, $n_2$ と入射角 $\theta_1$, 屈折角 $\theta_2$ の間には，式(7-11)に示すスネル(Snell)の法則が成り立ちます．この関係からベクトル $R$ (反射方向)，$T$ (屈折方向)は，下式のようになります．

$$n_1 \cdot \sin\theta_1 = n_2 \cdot \sin\theta_2 \qquad (7\text{-}11)$$

$$T = \frac{1}{n} \cdot \{L + (c-g) \cdot N\} \qquad (7\text{-}12)$$

$$R = L + 2 \cdot c \cdot N \qquad (7\text{-}13)$$
$$c = \cos\theta_1 = -L \cdot N$$
$$n = n_2 / n_1$$
$$g^2 = n^2 + c^2 - 1$$

図7-12　光の屈折

また，反射率 $k_r$，透過率 $k_t$ については，式7-14に示すフレーネル（Fresnel）の法則が成り立ちます．

$$k_r = \frac{1}{2} \cdot \left\{ \frac{\sin^2(\theta_1 - \theta_2)}{\sin^2(\theta_1 + \theta_2)} + \frac{\tan^2(\theta_1 - \theta_2)}{\tan^2(\theta_1 + \theta_2)} \right\} \qquad (7\text{-}14)$$

$$k_t + k_r = 1$$

上式を変形すると，反射率は下式のようになります．

$$k_r = \frac{1}{2} \cdot \left\{ \left( \frac{c-g}{c+g} \right)^2 + \left( \frac{n^2 \cdot c - g}{n^2 \cdot c + g} \right)^2 \right\} \tag{7-15}$$

透明で屈折する物体の描画について，考えてみましょう．図7-13に示すように視線$VR$が物体と交差する場合，反射や屈折が生ずるか判定します．反射や屈折が生ずる場合，交点$P_s$で視線が２つ（反射視線$RR$，屈折視線$TR$）にわかれます．わかれた視線についても，同様の操作を行い視線を追跡していく必要があります．そして，次の場合，追跡を打切ります．

・反射も透過もしない物体と視線が交差した．

・視線が物体と交差しない．

・反射や屈折が規定回数に達した．

その視線から見える色は，下式で計算します．

$$C = C_s + k_t \cdot C_t + k_r \cdot C_r \tag{7-16}$$

この式の計算では，図7-13に示す①の視線の処理を行う際に，②と③の視線の処理結果を必要としています．また，③の視線の処理を行う際には，同様に④と⑤の視線の処理結果を必要としています．したがって，この処理は再帰的なものとする必要があります．

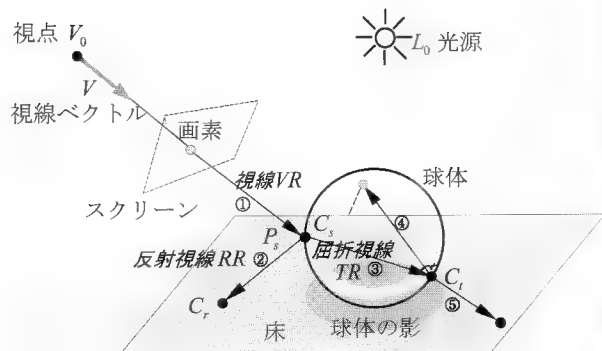



図7-13　球体と床の描画

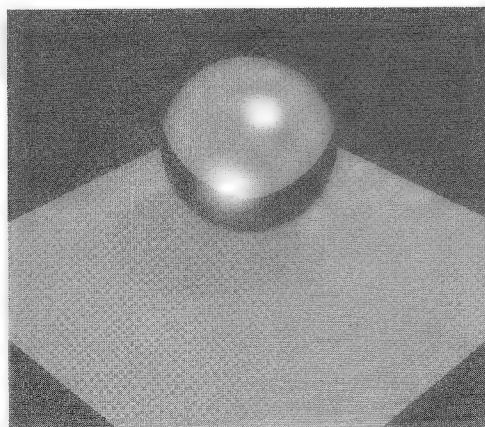



図7-14　屈折する物体(EX7_4.cpp)

透明で屈折する球体と床を描画するプログラム例(EX7_4.cpp)をリスト7-4に，実行結果を図7-14に示します．ヘッダファイルattri.h(巻末に示す)に物体の光学的性質を記述するために，Attriクラスを用意しています．

```
Attri(Color ic=WHITE, float id=0.7, float is=0.8, float ie=0.2,float ir=0, float in=0);
```

引数は順に，色，ディフューズ(拡散反射率)，スペキュラー(鏡面反射率)，アンビエント(環境光)，反射率，屈折率です．

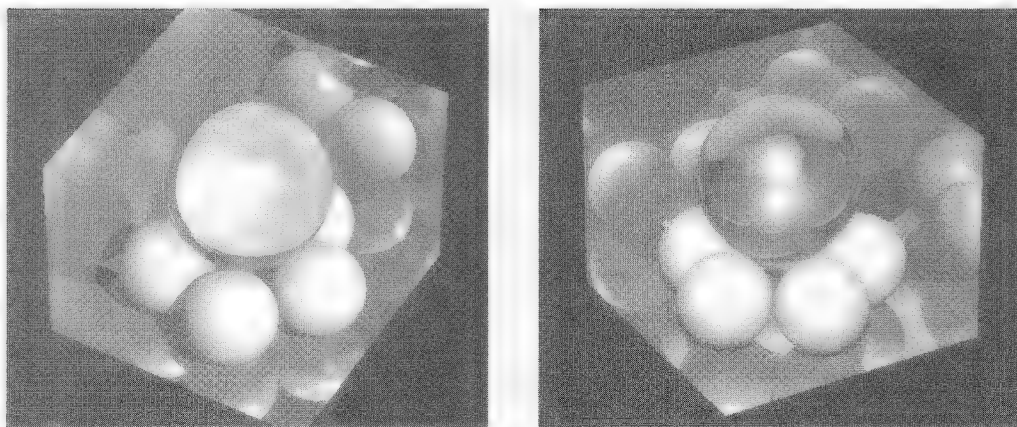



図7-15　積み重ねた球体(AP3_3.cpp)

また，前述の反射や屈折の処理のために，ヘッダファイルscene.h(巻末に示す)にSceneクラスを用意しています．前もって，物体をregist関数で登録しておき，color関数で前述の視線の処理を再帰的に実行してスクリーンの画素ごとに描画色を求めます．球体を積み重ねた場合のプログラム例(AP3_3.cpp)を，付録４に示します．

## 7.5　より自由な形状の表現

より自由な形状を表現する方法について考えてみましょう．球体と平面の描画を行ってきましたが，球体は下式に示す２次曲面の楕円面に属します．下式で表現できる２次曲面の形状を，図7-16に示します．

$$
\begin{aligned}
f(x,y,z) = & S_2 \cdot x \cdot x^2 + S_2 \cdot y \cdot y^2 + S_2 \cdot z \cdot z^2 \\
& + S_1 \cdot x \cdot x + S_1 \cdot y \cdot y + S_1 \cdot y \cdot z + S_0 = 0
\end{aligned}
\tag{7-17}
$$

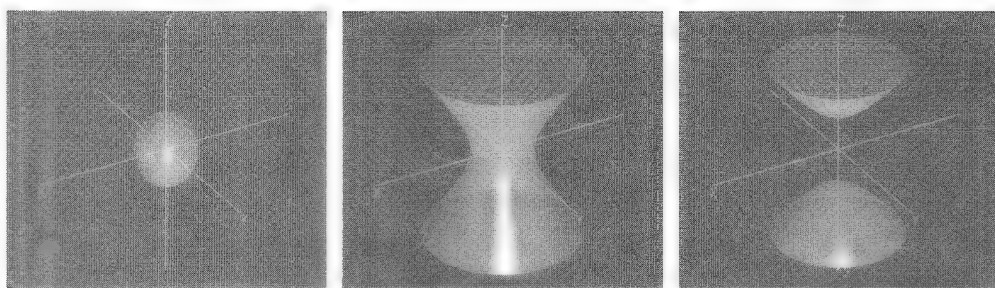
$S_2 = (S_2 \cdot x, S_2 \cdot y, S_2 \cdot z)$, $S_1 = (S_1 \cdot x, S_1 \cdot y, S_1 \cdot z)$, $S_0$ の値によって表現される曲面の代表的なものを，以下に示します．

①楕円面　②一葉双曲面　③二葉双曲面

図7-16　２次曲面(AP3_1.cpp)（その１）

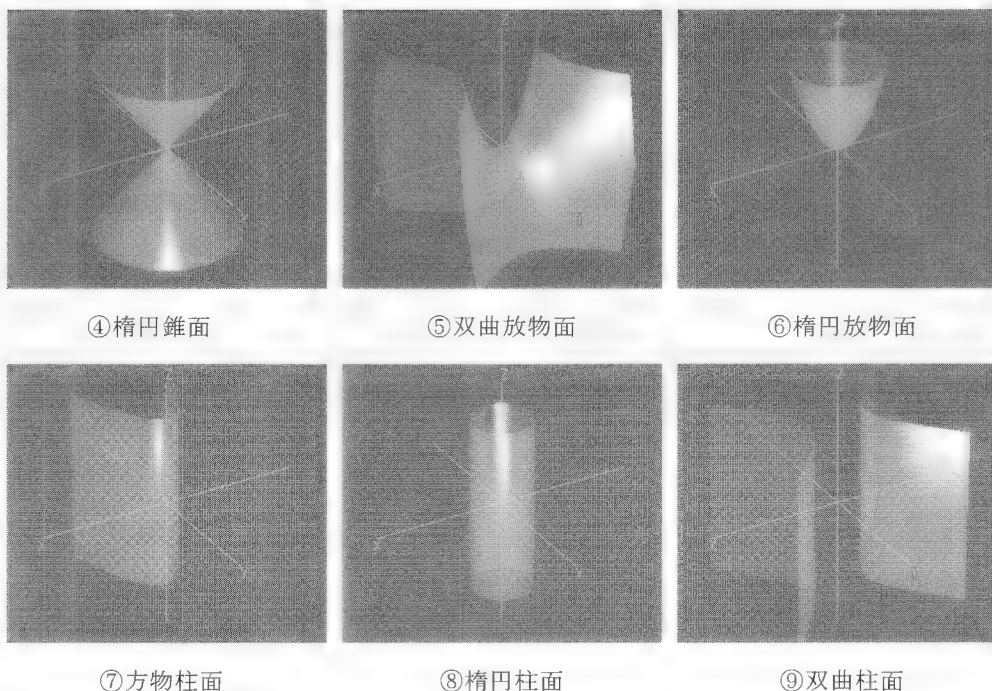



**図7-16**　２次曲面(AP3_1.cpp)（その２）

なお，上記①～⑨の各形状での$S_2, S_1, S_0$の値は，付録を参照して下さい．$S_2=(0,0,0)$の場合，上式は平面を表します．描画プログラムおよび，描画方法は付録を参照して下さい．

一般に，物体を描画するためには，以下の計算を行う必要があります．

・視線と物体との交点座標（交差判定を含む）

・交点での法線ベクトル

・物体の内部か外部かの判定

そのため，２次曲面を用いて立体を表現するために，$f(x,y,z)<0$の部分が立体の内部と考えることにします．２次曲面立体などの基本立体(primitive)は，上記の処理によって描画することができます．基本立体を用いて，より自由な形状の立体の描画について考えて行きましょう．

基本立体の間で集合演算に類似した立体の演算（和，差，積）を行うことによって，より自由な形状を表現するCSGモデル(constructive solid geometry)という表現方法があります．演算の内容は，以下のようになります．

立体の演算

| 演　算 | 記　述 | 演算結果の立体 |
|---|---|---|
| 和 | $a+b$ | 立体$a$または立体$b$に含まれる部分 |
| 差 | $a-b$ | 立体$a$に含まれ，かつ立体$b$に含まれない部分 |
| 積 | $a*b$ | 立体$a$および立体$b$の両方に含まれる部分 |

ＣＳＧモデルを記述するために，ヘッダファイルcsgm.h(巻末に示す)にCsgmクラスを用意しています．Csgmクラスを用いることによって，前述の２次曲面（$S_2,S_1,S_0$を指定）を使用することができます．また，上記の演算（和，積，差）や，座標変換(変換行列との積)を行うことで移動や拡大縮小ができます．演算の簡単なプログラム例(EX7_5.cpp)をリスト7-5に，実行結果を図7-17示します．

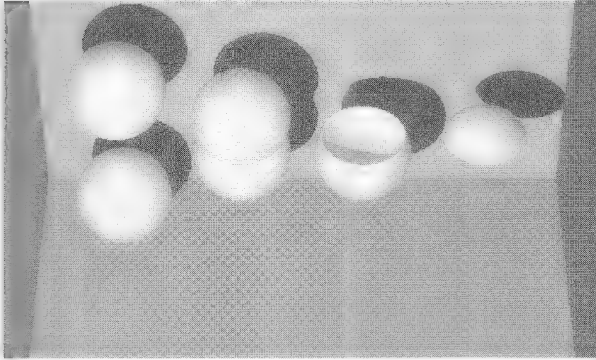



図7-17　立体の演算（和，差，積）(EX7_5.cpp)

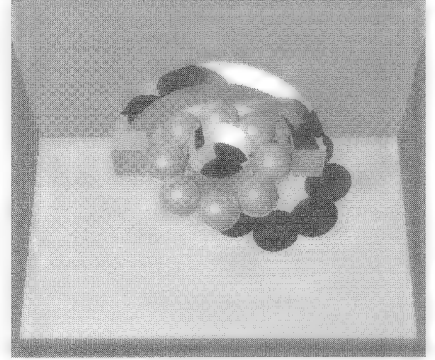



図7-18　ベアリングの断面形状(EX7_6.cpp)

ベアリングの断面形状を描画するプログラム例(EX7_6.cpp)をリスト7-6に，実行結果を図7-18示します．

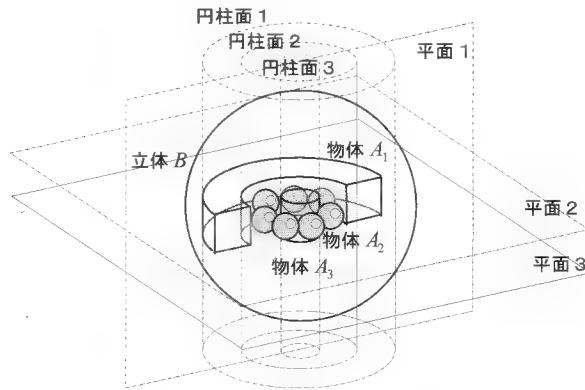



図7-19　バウンディングボリューム

レイトレーシングでは，EX7_6.cppの例のように物体が増えると視線との交点計算の回数が多くなり，計算時間が大きくなってきます．図7-19に示すように，CSGモデルで表される物体$A_1$は複数の面（平面1,平面2,平面3,円柱面1,円柱面2）を用いて記述されています．そのため，視線と各面との交点を求め，その交点が条件（立体の演算）を満たす場合に描画を行います．視線が物体$A_1$と交差しない場合（描画されない場合）でも交差判定のために，複数の面に対する交点計算が必要になります．また，物体$A_2$～$A_3$に対しても同様に交点計算が必要です．

交点計算の回数を削減して高速化する方法として，バウンディングボリューム(bounding volume)を用いる方法があります．物体$A_1$を包含する立体$B$（バウンディングボリューム，球体や直方体など)を定義します．まず，視線と立体$B$との交差判定を行い，交差する場合には物体$A_1$に対して本来の処理を行います．交差しない場合には，視線は物体$A_1$に交差しないので次の処理に移ることができます．結果的に，交点計算の回数が少なくなり，描画処理を高速化することができます．EX7_6.cppでは，$A_1$と$A_3$でバウンディングボリュームを使用しています．

より自由な形状を描画するために，メタボールとパラメトリック曲面を使用する方法があります．また，よりリアルな画像を生成する方法として，アンチエイリアシングやラジオシティ法などがあります．

（1）　メタボール(metaball)

人体などのように曲面の多い立体を，描画するのに適しています．その立体を球の集合として表します．各球は中心からの距離に応じて重み(濃度値)を持っており，その濃度値の関数は中心付近では大きな値となり，中心から離れるにしたがって小さくなり，やがて０となります．立体を構成する各球の濃度値の合計が一定値となる曲面を，立体として描画することによって，人体のようになめらかな立体とすることができます．２つの球を徐々に近づけた場合の例を口絵に示します．

（2）　パラメトリック曲面(parametric surface)

工業製品の形状のような自由曲面を，描画する場合に使用されます．複雑な立体の形状を，曲面の一部(曲面パッチ)を複数つなぎ合わせて構成します．その際に，パラメトリック曲面を使用します．パラメトリック曲面は，曲面を表す$x, y, z$座標値がそれぞれパラメータ$u$と$v$の関数（$x(u,v), y(u,v), z(u,v)$）となるものです．パラメトリック曲面の種類としては，クーンズ(Coons)曲面,ベジェ(Bezier)曲面，$B$-スプライン($B$-spline)曲面などがあります．コーヒーカップをパラメトリック曲面($B$スプライン)で描画するプログラム例を付録(AP4_2.cpp)に示します(口絵参照)．

（3）　アンチエイリアシング(antialiasing)

斜めの線分の描画は，２章で説明したように図2-2のように行うため，階段状のぎざぎざ(ジャギー:jaggy)が生じます．多角形の描画の際にも，各辺に対して同様に階段状のぎざぎざが生じます．このジャギーを軽減する処理をアンチエイリアシングといいます．ジャギーが発生する部分は，背景部分と描画部分の面積比を考慮して画素の色を決定します．アンチエイリアシングの例を，口絵28に示します．

（4）　ラジオシティ法(radiosity)

一般に着目している物体に届く光は，光源から光が直接届く直接光と，物体や壁などに一度当たってそこから反射した光が届く間接光からなります．レイトレーシング法は，直接光を考慮していますが，間接光や相互反射などは正確に表現できません．ラジオシティ法は，この光の相互反射を考慮することによって，影のぼやけ,間接光による照明などを表現することができ，よりリアルな画像を生成することができます．

## 演習問題

7-1　プログラム例EX7_3.cppを変更して，球体（映り込みのない）と映り込みのある床を描画せよ．

7-2　プログラム例EX7_3.cppおよびEX7_4.cppの反射率及屈折率を変更して，画像の変化を観察せよ．

7-3　プログラム例EX7_6.cppを下記のように変更して，バウンディングボリュームを使用しないで描画し，描画時間を比較せよ．

```
sc.regist( A1, Al, Ball(Point(0,0,25),110) );  ➔  sc.regist( A1, Al );
sc.regist( A3, Al, Ball(Point(0,0,25),40) );   ➔  sc.regist( A3, Al );
```

リスト 7-1　EX7_1.cpp

```
 1: //  球体を表示する(レイトレーシング法)
 2:
 3: #include        "graph3.h"
 4: #define         INFINITY    1e8                          //  無限遠方
 5:
 6: //  視線、光線のクラス
 7: class   Ray{
 8:         public:
 9:         Point   o;                                       //  始点座標
10:         Vector  d;                                       //  方向ベクトル
11:         Ray     (Point& or, Vector& di){    o=or;   d=di;   }
12: };
13:
14: //  球体のクラス
15: class   Ball{
16:         public:
17:         Point   o;                                       //  中心座標
18:         float   r;                                       //  半径
19:         Ball(Point& or=Point(0,0,0), float ra=0){   o=or; r=ra; }
20:         float   hit(Ray& VR, Point& P, Vector& N){       //  交点座標Pを返す)
21:                 float   ds=VR.d*(VR.o-o);
22:                 float   d2=ds*ds-(VR.o-o)*(VR.o-o)+r*r; //  判別式
23:                 if( d2 < 0 )    return INFINITY;
24:                 float   t=-ds-sqrt(d2);                  //  距離
25:                 P=VR.o+t*VR.d;                           //  交点座標
26:                 N=unit(P-o);                             //  法線ベクトル
27:                 return t;
28:         }
29: };
30:
31: //  反射光を計算する    V:視線ベクトル  L:照明方向ベクトル  N:法線ベクトル s:照明光強度
32: Color   shading(Vector& V, Vector& L, Vector& N, Color& c,float s=1){
33:         float   kd=0.7, ks=0.7, ke=0.3;                  //拡散反射係数,鏡面反射係数,環境光
34:         Vector  R=L-2*(L*N)*N;                           //  反射方向ベクトル
35:         return  kd*max(-N*L,0)*s*c+ks*pow(max(-R*V,0),20)*s*WHITE+ke*c;
36: }
37:
38: main()
39: {
40:     Ball    Ba= Ball( Point(0,0,70), 70 );               //  球体の記述
41:     Point   L0=Point( 0, 400, 1000 );                    //  照明光の記述
42:     Point   V0=Point( 500, 500, 500 );                   //  視点の記述
43:     Win3D   w("EX7_1", 0.25*CYAN);
44:     w.setview(V0);
45:
46:     for(int x=0; x<w.size_x; x++)
47:         for(int y=0; y<w.size_y; y++){
48:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);// 視線ベクトルの計算
49:             Ray     VR=Ray(V0,V);                        //  視線の記述
50:             Point   P;  Vector  N;                       //  交点座標
51:             if( Ba.hit(VR,P,N)<INFINITY ){               //  視線が球体と交わるかの判定
52:                 Vector  L=unit(P-L0);                    //  照明光の方向ベクトル
53:                 w.color( shading(V,L,N,RED) );
54:                 w.point(x,y);
55:             }
56:     }
57:     pause("EX7_1");
58: }
```

リスト 7-2　EX7_2.cpp

```
 1: //  球体と床を表示する(レイトレーシング法)
 2:
 3: #include        "graph3.h"
```

```
 4: #include        "ray.h"
 5:
 6: main()
 7: {
 8:     Ball    Ba= Ball( Point(0,0,70),70);                   //  球体の記述
 9:     Polygon Wp( Point( 200,  200,-50),Point(-200,  200,-50),//    床の記述
10:                 Point(-200,-200,-50),Point( 200,-200,-50) );
11:     Wall    Wa= Wall( Wp );
12:     Color   Cb= RED;
13:     Color   Cw= 0.3*WHITE+0.4*BLUE;
14:
15:     Win3D   w("EX7_2");
16:     Point   V0=Point(500,500,500);                         //  視点の記述
17:     Point   L0=Point(0,400,1000);                          //  照明光の記述
18:     w.setview(V0);
19:
20:     for(int x=0; x<w.size_x; x++)
21:         for(int y=0; y<w.size_y; y++){
22:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);// 視線ベクトルの計算
23:             Ray     VR=Ray(V0,V);                          //  視線の記述
24:             float   s=1;                                   //  日向か日陰を表す
25:
26:             Point   P;                                     //  交点座標
27:             Vector  N,D;
28:             Color   Cs=0.25*CYAN;
29:             if(Ba.hit(VR,P,N)<INFINITY){                   //  視線が球体と交わるかの判定
30:                 Vector  L=unit(P-L0);                      //  照明光の方向ベクトル
31:                 Cs=shading(V,L,N,Cb,1);
32:             }
33:             else if(Wa.hit(VR,P,N)<INFINITY){              //  視線が床と交わるかの判定
34:                 Vector  L=unit(P-L0);                      //  照明光の方向ベクトル
35:                 Ray     LR=Ray(L0, L);                     //  照明光線の記述
36:                 if(Ba.hit(LR,P,D)<INFINITY)s=0;            //  照明光が球体と交わるかの判定
37:                 Cs=shading(V,L,N,Cw,s);
38:             }
39:             w.color(Cs);
40:             w.point(x,y);
41:     }
42:     pause("EX7_2");
43: }
```

リスト 7-3　EX7_3.cpp

```
 1: //  レイトレーシング法
 2: //  床と映り込みのある球体を表示する
 3:
 4: #include        "graph3.h"
 5: #include        "ray.h"
 6:
 7: main()
 8: {
 9:     Ball    Ba= Ball( Point(0,0,70),70);                   //  球体の記述
10:     Polygon Wp( Point( 200,  200,-50),Point(-200,  200,-50),
11:                 Point(-200,-200,-50),Point( 200,-200,-50) );
12:
13:     Wall    Wa= Wall( Wp );                                //  床の記述
14:     Color   Cb= RED;
15:     Color   Cw= 0.3*WHITE+0.4*BLUE;
16:     float   kr=0.3;                                        //  反射率
17:
18:     Win3D   w("EX7_3");
19:     Point   V0=Point(500,500,500);                         //  視点の記述
20:     Point   L0=Point(0,400,1000);                          //  照明光の記述
21:     w.setview(V0);
22:
23:     for(int x=0; x<w.size_x; x++)
```

```
24:         for(int y=0; y<w.size_y; y++){
25:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);// 視線ベクトルの計算
26:             Ray     VR=Ray(V0,V);                        // 視線の記述
27:             float   s=1;                                 // 日向か日陰を表す
28:
29:             Point   Ps,Pr;                               // 交点座標
30:             Vector  N,D;
31:             Color   Cs=0.25*CYAN,Cr;
32:             if(Ba.hit(VR,Ps,N)<INFINITY){                // 視線が球体と交わるかの判定
33:                 Vector  L=unit(Ps-L0);                   // 照明光の方向ベクトル
34:                 Cs=shading(V,L,N,Cb,1);
35:                 Ray     RR=Ray(Ps, V-2*(V*N)*N);         // 反射視線の記述
36:                 if(Wa.hit(RR,Pr,N)<INFINITY){            // 視線が床と交わるかの判定
37:                     Vector  Lr=unit(Pr-L0);              // 照明光の方向ベクトル
38:                     Ray     LR=Ray(L0, Lr);              // 照明光線の記述
39:                     if(Ba.hit(LR,Pr,D)<INFINITY)s=0;     // 照明光が床に当たるかの判定
40:                     Cr=shading(RR.d,Lr,N,Cw,s);
41:                 }
42:             }
43:             else if(Wa.hit(VR,Ps,N)<INFINITY){           // 視線が床と交わるかの判定
44:                 Vector  L=unit(Ps-L0);                   // 照明光の方向ベクトル
45:                 Ray     LR=Ray(L0, L);                   // 照明光線の記述
46:                 if(Ba.hit(LR,Ps,D)<INFINITY)s=0;         // 照明光が床に当たるかかの判定
47:                 Cs=shading(V,L,N,Cw,s);
48:             }
49:             w.color(Cs+kr*Cr);
50:             w.point(x,y);
51:     }
52:     pause("EX7_3");
53: }
```

リスト 7-4　EX7_4.cpp

```
 1: //  物体（球体、床）を複数にする（影,多重反射,屈折の処理も行う）
 2:
 3: #include        "graph3.h"
 4: #include        "scene.h"
 5:
 6: #define         NUM_REF 6                                // 多重反射の回数
 7: main()
 8: {
 9:     Scene   sc(0.1*CYAN);                                // 複数物体（球体、壁）の記述
10:     Attri   Ac=Attri(BLACK, 0.7, 0.7, 0.3, 0.2,1.3);     // 球体の性質の記述
11:             //          色,拡散,鏡面,環境,反射,屈折率
12:     sc.regist ( Ball(Point(0,0,70),70), Ac );            // 球体の記述
13:     Polygon Wp( Point( 200, 200,-50),Point(-200, 200,-50),
14:                 Point(-200,-200,-50),Point( 200,-200,-50) );
15:     sc.regist ( Wall(Wp), Attri(0.3*WHITE+0.4*BLUE) );   // 床の記述
16:
17:     Win3D   w0("EX7_4");
18:     Point   V0=Point(500,500,500);                       // 視点の記述
19:     Point   L0=Point(0,400,1000);                        // 照明光の記述
20:     w0.setview(V0);
21:     for(int x=0,n; x<w0.size_x; x++)                     // スクリーン座標x,yの描画
22:         for(int y=0; y<w0.size_y; y++){
23:             Vector  V=unit(w0.world(Point(x,y,-w0.dv))-V0);//   視線ベクトル
24:             Ray     VR=Ray(V0,V);                        // 視線の記述
25:             w0.color( sc.color(VR,L0,NUM_REF) );
26:             w0.point(x,y);
27:         }
28:     pause("EX7_4");
29: }
```

リスト 7-5 EX7_5.cpp

```
 1: //  球体の演算（CSG モデル）
 2:
 3: #include        "graph3.h"
 4: #include        "scene.h"
 5:
 6: #define         NUM_REF 6                               //  視線衝突・通過制限(>0)
 7:
 8: main()
 9: {
10:     Scene   sc(0.25*CYAN);                          //  複数物体（球体、壁）の記述
11:
12:     //  物体の記述
13:     Attri   AI=Attri(YELLOW,0.7,0.7,0.3, 0.15 );
14:
15:     Csgm    C1=move(0,0,-20)*ball(Point(40,40,40));     //  半径 40 の球体
16:     Csgm    C2=move(0,0, 20)*ball(Point(40,40,40));     //  半径 40 の球体
17:
18:     sc.regist( move( 150,0, 0)*C1 , AI );             //  球体の登録
19:     sc.regist( move( 150,0,80)*C2 , AI );             //  球体の登録
20:     sc.regist( move(  50,0,50)*(C1+C2) , AI );        //  球体の和
21:     sc.regist( move( -50,0,50)*(C1-C2) , AI );        //  球体の減算
22:     sc.regist( move(-150,0,50)*(C1*C2) , AI );        //  球体の積
23:
24:     //  床、壁の記述（Wa：ポリゴン）
25:     #define WID     230
26:     #define HEI     120
27:     Wall    wa1(Polygon(Point(-WID, HEI,-HEI),Point(-WID,-HEI,-HEI),
28:                         Point( WID,-HEI,-HEI),Point( WID, HEI,-HEI)) );
29:     Wall    wa2(Polygon(Point(-WID,-HEI,-HEI),Point(-WID,-HEI, HEI),
30:                         Point( WID,-HEI, HEI),Point( WID,-HEI,-HEI)) );
31:
32:     sc.regist( wa1, Attri(0.7*WHITE+0.3*BLUE));
33:     sc.regist( wa2, Attri(0.7*WHITE+0.3*BLUE));
34:
35:     //  描画
36:     Win3D   w1("EX7_5w1",500,300),w2("EX7_5w2",500,300);
37:     w1.setview( 90,45,2000);
38:     w2.setview( 60,20,2000);
39:     Point   V1=w1.v0;                               //  視点 0 の記述
40:     Point   V2=w2.v0;                               //  視点 1 の記述
41:     Point   L0=Point(200,1000,600);                 //  照明光の記述
42:
43:     for(int x=0,n; x<w1.size_x; x++)                  //  スクリーン座標 x,y の描画
44:         for(int y=0; y<w1.size_y; y++){
45:             Vector  V=unit(w1.world(Point(x,y,-w1.dv))-V1);//   視線ベクトル
46:             Ray     VR=Ray(V1,V);                   //  視線の記述
47:             w1.color( sc.color(VR,L0,NUM_REF) );
48:             w1.point(x,y);
49:
50:             V=unit(w2.world(Point(x,y,-w2.dv))-V2);     //  視線ベクトル
51:             VR=Ray(V2,V);                           //  視線の記述
52:             w2.color( sc.color(VR,L0,NUM_REF) );
53:             w2.point(x,y);
54:         }
55:     pause("EX7_5");
56: }
```

リスト 7-6 EX7_6.cpp

```
 1: //  ベアリングのカットモデルの描画
 2:
 3: #include        "graph3.h"
 4: #include        "scene.h"
 5: #define         NUM_REF 20                              //  視線衝突・通過制限(>0)
```

```
 6:
 7: main()
 8: {
 9:     Scene   sc(0.25*WHITE);                              // 複数物体（球体、壁）の記述
10:     Attri   Al=Attri(BLACK, .7,.7,.3,  .1,1.3,.95);     // 屈折率 1.5
11:     Attri   At=Attri(.3*WHITE,.7,.9,.3, .5);
12:
13:     Csgm    C1=cylinder(Point(100,100,0));               // 半径 100 の円筒(円柱面 1)
14:     Csgm    C2=cylinder(Point( 70, 70,0));               // 半径  70 の円筒(円柱面 2)
15:     Csgm    P1=plane_xz();                               // XZ 平面(平面 1)
16:     Csgm    P2=move(0,0,50)*plane_xy();                  // XY 平面(平面 2)
17:     Csgm    P3=plane_xy();                               // XY 平面(平面 3)
18:     Csgm    A1=(C1-C2)*P1*P2*(-P3);                      // シリンダの外枠(物体 A1)
19:     sc.regist( A1, Al, Ball(Point(0,0,25),110) );
20:
21:     Point   A2=Point(70-20,0,25);
22:     for(int i=0; i<8; i++)
23:         sc.regist( Ball(rot_z(45*i)*A2,20), At );        // ベアリングの球(物体 A2)
24:
25:     Csgm    C3=cylinder(Point( 30, 30, 0));              // 半径  30 の円筒(円柱面 3)
26:     Csgm    A3=C3*P2*(-P3);                              // ベアリングの回転軸(物体 A3)
27:     sc.regist( A3, Al, Ball(Point(0,0,25),40) );
28:
29:     // 床、壁の記述 (Wa：ポリゴン)
30:     #define WID     200
31:     #define HEI     120
32:     Wall    wa1(Polygon(Point( WID, HEI,-HEI),Point(-WID, HEI,-HEI),
33:                         Point(-WID,-HEI,-HEI),Point( WID,-HEI,-HEI)) );
34:     Wall    wa2(Polygon(Point(-WID,-HEI,-HEI),Point(-WID,-HEI, HEI),
35:                         Point( WID,-HEI, HEI),Point( WID,-HEI,-HEI)) );
36:     sc.regist( wa1, Attri(0.7*WHITE+0.3*BLUE));
37:     sc.regist( wa2, Attri(0.7*WHITE+0.3*BLUE));
38:
39:     // 描画
40:     Win3D   w1("EX7_6w1"),w2("EX7_6w2");
41:     w1.setview( 90,60,1000);
42:     w2.setview( 60,45,1000);
43:     Point   V1=w1.v0;                                    // 視点 0 の記述
44:     Point   V2=w2.v0;                                    // 視点 1 の記述
45:     Point   L0=Point(1000*unit(45,60));                  // 照明光の記述
46:
47:     for(int x=0,n; x<w1.size_x; x++)                     // スクリーン座標 x,y の描画
48:         for(int y=0; y<w1.size_y; y++){
49:             Vector  V=unit(w1.world(Point(x,y,-w1.dv))-V1); //  視線ベクトル
50:             Ray     VR=Ray(V1,V);                        // 視線の記述
51:             w1.color( sc.color(VR,L0,NUM_REF) );
52:             w1.point(x,y);
53:
54:             V=unit(w2.world(Point(x,y,-w2.dv))-V2);      // 視線ベクトル
55:             VR=Ray(V2,V);                                // 視線の記述
56:             w2.color( sc.color(VR,L0,NUM_REF) );
57:             w2.point(x,y);
58:         }
59:     pause("EX7_6");
60: }
61:
```

# 8．マッピング

この章では，物体の表面に模様や小さなでこぼこによる質感の変化，および周囲の景色の映り込みや屈折の実現方法について考えていきます．表面に絵や模様を貼り付けることをテクスチャマッピング(texture mapping)といいます．また，表面に凹凸をつけることをバンプマッピング(bump mapping)といいます．さらに，周囲の景色を物体表面に反映（反射，屈折）することを環境マッピングといいます．

上記の各マッピングを行うためには，平面の画像（マッピング画像）を物体表面（平面とは限らない）にどのように対応させるかを考える必要があります．すなわち，物体表面上の点（座標）に対して，画像のどこ（画像上の座標）を対応させる（マッピング）かという関数が必要となります．下図に示すように，画像を球状にマッピングする，画像を円筒状に丸めてマッピングする，画像を平行にマッピングする方法などがあります（①と②は，まず平面の画像を球体あるいは円筒にマッピングを行い，次に下図のようにマッピングを行うので，２段階のマッピングになります）．

① 球状にマッピング

② 円筒状にマッピング

③ 平行にマッピング

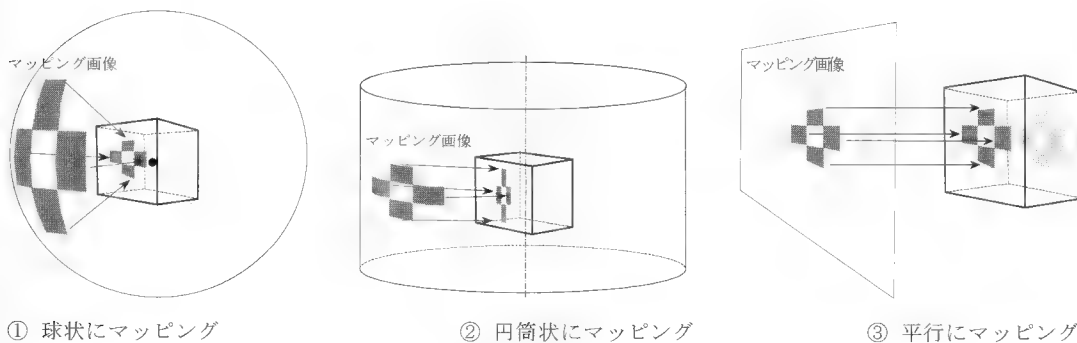



図8-1　マッピング

## 8．1　テクスチャマッピング

中心が原点にある球体の表面に，絵や模様の画像を貼り付ける方法について考えます．球体を地球に例えると，貼り付ける画像の上端と下端をそれぞれ北極と南極の位置に対応させ，赤道上には画像の中間の高さ部分が対応するようにします．

図8-2に示すように，レイトレーシング法を用いた場合，視線が球体と交わる点Pでの法線ベクトルを$N$とし，点Pに対応するマッピング画像の座標（$X'Y'$座標）を$M$とします．法線ベクトル$N$と画像座標$M$との対応関係は次式のようになります（ただし，$r$は球体の半径であり，Pは各点の座標を表すとともに原点からのベクトルを表すものとする）．北極と南極部分は画像の歪みが大きくなりますが，赤道付近はあまり歪みません．

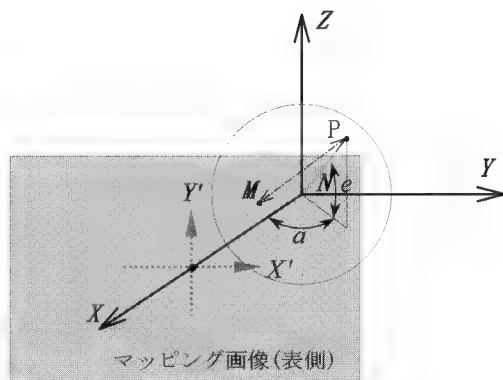



**図8-2**　テクスチャマッピング

$$N = \frac{P}{r}$$　法線ベクトル　(8-1)

$$a = \arccos\left(\frac{N_x}{\sqrt{{N_x}^2 + {N_y}^2}}\right)$$　$0 \le a \le \pi$ （$N_y \ge 0$ の場合）　(8-2)

$$a = -\arccos\left(\frac{N_x}{\sqrt{{N_x}^2 + {N_y}^2}}\right)$$　$-\pi < a < 0$ （$N_y < 0$ の場合）

$$e = \arcsin(N_z)$$　$-\frac{\pi}{2}\pi \le e \le \frac{\pi}{2}$

$$M_x = \frac{a \cdot w_i}{2\pi}$$　$w_i$ ：マッピング画像の横幅　(8-3)

$$M_y = \frac{e \cdot h_i}{\pi}$$　$h_i$ ：マッピング画像の高さ

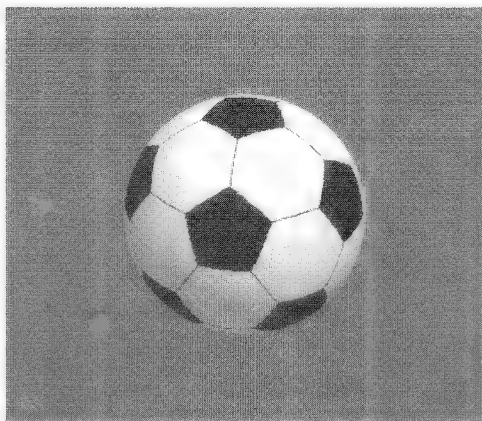
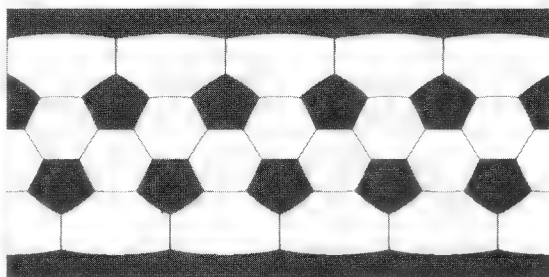



**図8-3**　テクスチャマッピング(EX8_1.cpp)

(texture.bmp)

上記の関係でテクスチャマッピングするプログラム例(EX8_1.cpp)をリスト8-1に，実行結果を図8-3に示します．実行するとウィンドウが２つ表示されます．最初のウィンドウには，マッピング画像が表示されます．次のウィンドウには，テクスチャマッピングされた球体が表示されます．プログラム例では，上式のマッピング画像(Win2Dクラス)の座標を求める計算は，mapping関数を用意して実行しています（mapping関数では $M_x = \mathrm{dir} \cdot a \cdot w_i / 2\pi$ と計算しているため，引数を $\mathrm{dir} = 1$ (省略値)としている）．

画像を物体の表面に貼り付けるだけでなく，金太郎飴のように物体の内部まで浸透させることを，ソリッドテクスチャリング（solid texturing）といいます．図8-4および下式に示すように，球体上の点Pに対応するテクスチャ画像の位置 $M$ は，点Pの $Y$ 座標と $Z$ 座標の値によって決まり，$X$ 座標は関係しません．したがって，球体内部まで $X$ 軸方向に平行にマッピングされます（図8-1 ③を参照）．

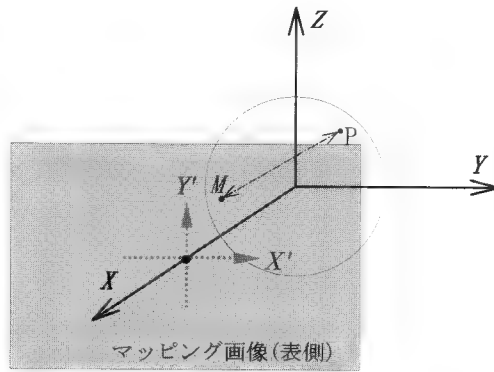



図8-4 ソリッドテクスチャリング

$$M_x = P_y \qquad r：球体の半径 \qquad (8-4)$$
$$M_y = P_z$$

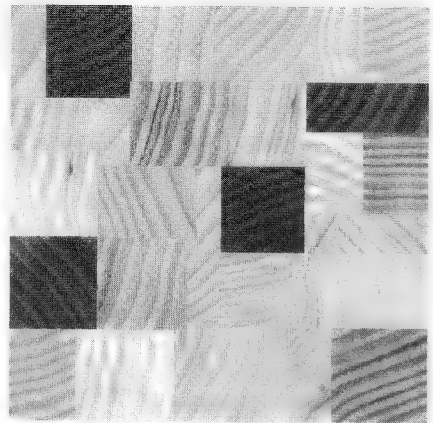
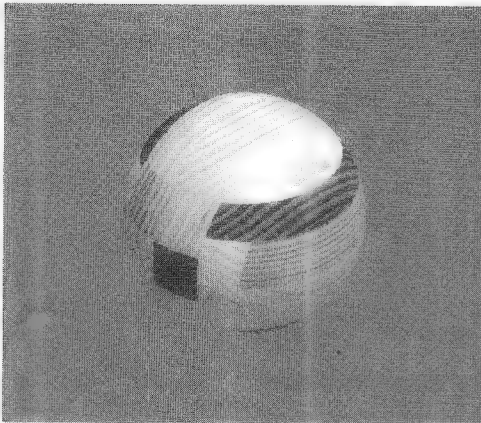



図8-5 ソリッドテクスチャリング(EX8_2.cpp)

(wood.bmp)

上記の関係でテクスチャマッピングするプログラム例(EX8_2.cpp)をリスト8-2に，実行結果を図8-5に示します．実行するとウィンドウが2つ表示されます．最初のウィンドウには，マッピング画像が表示されます．次のウィンドウには，ソリッドテクスチャリングされた球体が表示されます．

## 8.2　バンプマッピング

物体の表面に小さな凹凸を付ける方法として，バンプマッピングがあります．物体表面の凹凸に対応する画像によって，シェーディング処理に用いる物体表面の法線ベクトルを変更して，疑似的に凹凸感を出します．図8-6に示すように，凹凸を表すマッピング画像（モノクロ画像）から法線ベクトル $F$ を生成します．そして，物体表面の法線ベクトル $N$ と合成して，シェーディング（明暗を付ける）に用いる法線ベクトル $B$ を計算します．物体表面の形状を変更して凹凸を付けているわけではありませんが，シェーディングの結果として明暗がつき，擬似的に凹凸感を出すことができます．

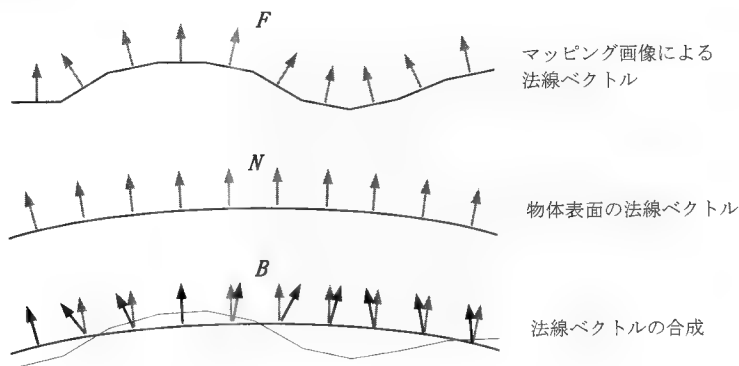



**図8-6**　法線ベクトルの合成

マッピング画像から，その法線ベクトル $F$ を求める方法を示します．図8-7に示すように，マッピング画像（モノクロ画像）の各画素の輝度レベル $l$ は，その座標（位置）での高さ（標高値）を表しているものとして，凹凸（地形）を考えます．その表面の法線ベクトル $F$ は，表面に接する $X'$ 方向のベクトル $DX$ および $Y'$ 方向のベクトル $DY$ から下式で求めることができます．

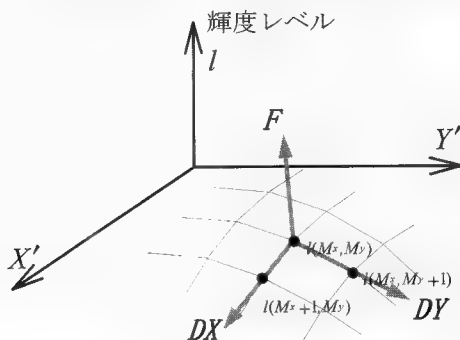



**図8-7**　法線ベクトル

$$F = \mathrm{rot_}z(90) * \frac{DX \times DY}{|DX \times DY|}$$

$$DX = (1,0,l(M_x+1,M_y)-l(M_x,M_y)) \qquad (8\text{-}5)$$

$$DY = (0,1,l(M_x,M_y+1)-l(M_x,M_y))$$

上式で$l$軸は$Z$軸に対応するので，傾斜がない場合にはベクトル$F$は（0,0,1）となります．したがって，ベクトル(0, 0, 1)からベクトル$F$へ回転する変換（rotate(Point(0,0,1),F)と記述できる）を，ベクトル$N$に施すことによってベクトル$B$を合成することができます．プログラム上では以下のように記述できます．

```
B=rotate(Point(0,0,1),F)*N;
```

物体表面の位置$P$とマッピング画像との対応は，テクスチャマッピングの場合と同じであり，mapping関数で実行しています．

上記の関係でバンプマッピングするプログラム例(EX8_3.cpp)をリスト8-3に，実行結果を図8-8に示します．実行するとウィンドウが2つ表示されます．最初のウィンドウには，マッピング画像が表示されます．次のウィンドウには，バンプマッピングされた球体が表示されます．

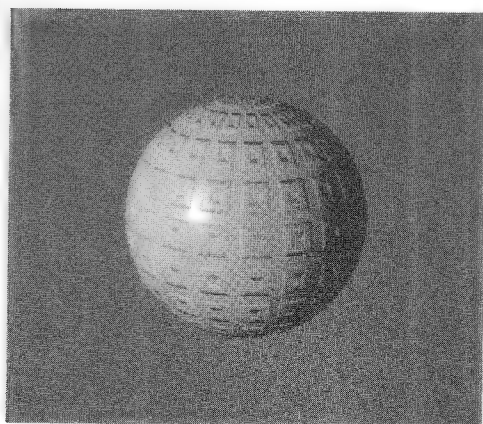
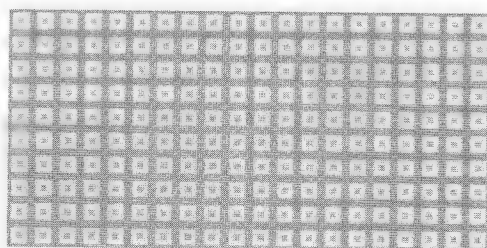



図8-8　バンプマッピング（EX8-3.cpp）

(bump.bmp)

## 8.3　環境マッピング

物体を描画する場合，反射や屈折を考えると，その周囲の別の物体の情報が必要になる場合があります．しかし，周囲の状況（物体の配置など）が複雑になってくると，すべてを考慮することは難しくなってきます．そこで，周囲の状況（環境）を擬似的に背景画像（マッピング画像）として，物体の描画に使用する方法が環境マッピングです．反射に適用することを，リフレクションマッピング（reflection mapping）といいます．また，透明な物体の屈折に適用することを，リフラクションマッピング（refraction mapping）といいます．マッピング画像は，描画する物体を取り巻く大きな球，円筒，六面体などとして利用します．

（1）　リフレクションマッピング

反射する球体を描画する場合に，リフレクションマッピングを行う方法について考えます．図8-9に示すように，描画する球体の周りにテクスチャマッピングした半径の大きな仮想球を考えます．このとき，反射視線から見える仮想球の色（マッピング画像のM点の色$C_r$）を求めます．

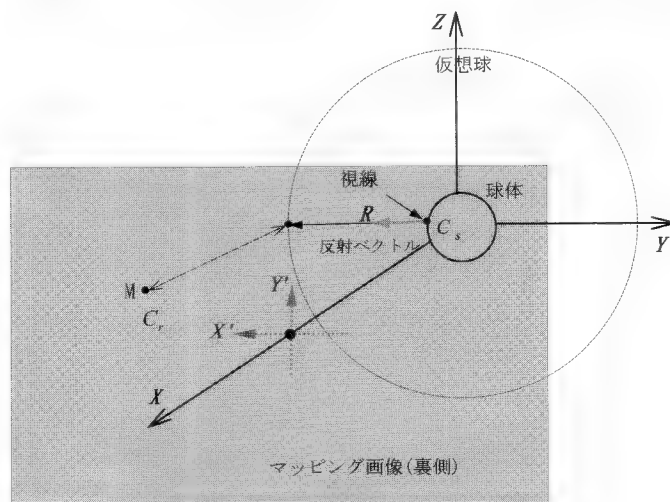



図8-9　リフレクションマッピング

下式に，反射ベクトル$R$とマッピング画像の座標$M$との関係を示します．この関係は，テクスチャマッピングの場合の物体の法線ベクトルとマッピング画像の座標との関係と似ていますが，マッピング画像の裏表が逆になっているので，$M_x$の式が異なります．

$$a = \arccos\left(\frac{R_x}{\sqrt{R_x{}^2 + R_y{}^2}}\right) \qquad 0 \le a \le \pi \quad （R_y \ge 0\ の場合） \qquad (8\text{-}6)$$

$$a = -\arccos\left(\frac{Rx}{\sqrt{R_x{}^2 + R_y{}^2}}\right) \qquad -\pi < a < 0\ （R_y < 0\ の場合）$$

$$e = \arcsin(R_z) \qquad -\frac{\pi}{2}\pi \le e \le \frac{\pi}{2}$$

$$M_x = -\frac{aw_i}{2\pi} \qquad w_i：マッピング画像の横幅 \qquad (8\text{-}7)$$

$$M_y = \frac{eh_i}{\pi} \qquad h_i：マッピング画像の高さ$$

上記の関係でリフレクションマッピングを行うプログラム例（EX8_4.cpp）をリスト8-4に，実行結果を図8-10に示します．実行するとウィンドウが２つ表示されます．最初のウィンドウには，マッピング画像が表示されます．次のウィンドウには，リフレクションマッピングされた球体が表示されます．プログラム例では，上式のマッピング画像の座標を求める計算は，mapping

[illegible]を用意して実行しています（mapping関数では $M_x = \mathrm{dir} \cdot a \cdot w_i / 2\pi$ と計算しているため，[illegible]数をdir＝−1としている）．

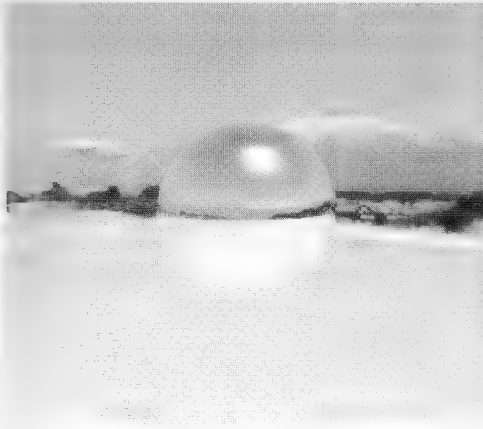



図8-10 リフレクションマッピン（EX8_4.cpp） （snow.bmp）

（2） リフラクションマッピング

透明で屈折する球体を描画する場合に，リフラクションマッピングを行う方法について考えます．リフレクションマッピングの場合と同様に，下図に示すように描画する球体の周りにテクスチャマッピングした半径の大きな仮想球を考え，透過視線から仮想球の色（マッピング画像のM点の色 $C_r$）を求めます．

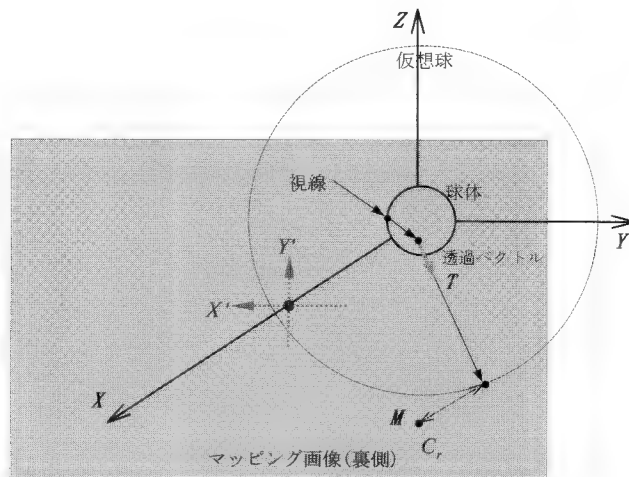



図8-11 リフラクションマッピング

透過ベクトル $T$ とマッピング画像の座標 $M$ との関係は，リフレクションマッピングの場合の物体の反射ベクトルとマッピング画像の座標との関係と同様に考えることができます．

リフラクションマッピングを行うプログラム例(EX8_5.cpp)をリスト8-5に，実行結果を図8-12に示します．実行するとウィンドウが２つ表示されます．最初のウィンドウには，マッピング画像が表示されます．次のウィンドウには，リフラクションマッピングされた球体が表示されます．

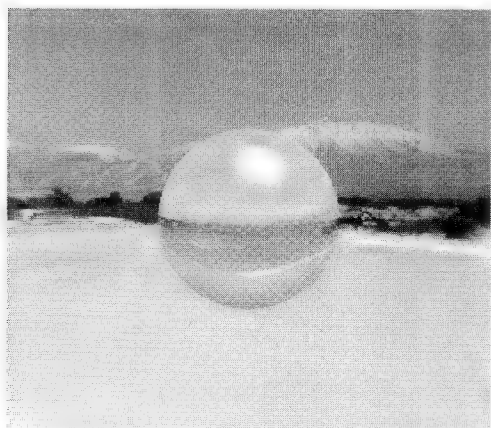



図8-12　リフラクションマッピング(EX8_5.cpp)　　(snow.bmp)

## 演習問題

8-1　プログラム例EX8_3.cppを変更して，球体にテクスチャマッピングとバンプマッピングを行って描画せよ．マッピング画像は，"..¥¥pic¥¥texture2.bmp"，"..¥¥pic¥¥bump2.bmp"を使用せよ．

リスト 8-1 EX8_1.cpp

```
 1: //  球体にテクスチャマッピングする
 2:
 3: #include        "graph1.h"
 4: #include        "graph3.h"
 5: #include        "ray.h"
 6:
 7: //  マッピング座標を返す
 8: Point   mapping(Win2D& w, Vector& V, int
 9:         float   a=0,        e=asin(V.z);                //  a:方位角 e:仰角
10:         int     wi=w.size_x,hi=w.size_y;
11:         if      (V.y>0)  a= acos(V.x/sqrt(V.x*V.x+V.y*V.y));
12:         else if(V.y<0)  a=-acos(V.x/sqrt(V.x*V.x+V.y*V.y));
13:         return  Point( (int)(dir*a*wi/PI/2)%wi, e*hi/PI );
14: }
15:
16: main()
17: {
18:     Ball    Ba= Ball( Point(0,0,0),100);                //  球体の記述
19:
20:     Win2D   tex("","..¥¥pic¥¥texture.bmp");             //  テクスチャ画像用ウィンドウ
21:
22:     Win3D   w("EX8_1",0.25*CYAN);
23:     Point   V0=Point(500,500,500);                      //  視点の記述
24:     Point   L0=Point(0,400,1000);                       //  照明光の記述
25:     w.setview(V0);
26:
27:     for(int x=0; x<w.size_x; x++)
28:         for(int y=0; y<w.size_y; y++){
29:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);// 視線ベクトルの計算
30:             Ray     VR=Ray(V0,V);                       //  視線の記述
31:
32:             Point   Ps;                                 //  交点座標
33:             Vector  N;                                  //  法線ベクトル
34:             if(Ba.hit(VR,Ps,N)<INFINITY){               //  視線が球体と交わるかの判定
35:                 Vector  L=unit(Ps-L0);                  //  照明光の方向ベクトル
36:
37:                 Point   M= mapping(tex,N);              //  マッピング画像の座標
38:                 Color   c= tex.pixel(M);                //  マッピング画像の色
39:                 w.color( shading(V,L,N,c) );
40:                 w.point(x,y);
41:             }
42:     }
43:     pause("EX8_1");
44: }
```

リスト 8-2 EX8_2.cpp

```
 1: //  球体にテクスチャマッピングする
 2:
 3: #include        "graph1.h"
 4: #include        "graph3.h"
 5: #include        "ray.h"
 6:
 7: main()
 8: {
 9:     Ball    Ba= Ball( Point(0,0,0),99);                 //  球体の記述
10:
11:     Win2D   tex("","..¥¥pic¥¥wood.bmp");                //  テクスチャ画像用ウィンドウ
12:
13:     Win3D   w("EX8_2", 0.25*CYAN);
14:     Point   V0=Point(500,500,500);                      //  視点の記述
15:     Point   L0=Point(0,400,1000);                       //  照明光の記述
16:     w.setview(V0);
17:
```

```
18:     for(int x=0; x<w.size_x; x++)
19:         for(int y=0; y<w.size_y; y++){
20:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);// 視線ベクトルの計算
21:             Ray     VR=Ray(V0,V);                         // 視線の記述
22:
23:             Point   Ps;                                   // 交点座標
24:             Vector  N;                                    // 法線ベクトル
25:             if(Ba.hit(VR,Ps,N)<INFINITY){                 // 視線が球体と交わるかの判定
26:                 Vector  L=unit(Ps-L0);                    // 照明光の方向ベクトル
27:
28:                 Point   C=99*Point(N.y,N.z);              // テクスチャ画像の座標計算
29:                 Color   c=tex.pixel(C);                   // テクスチャ画像の色
30:                 w.color( shading(V,L,N,c) );
31:                 w.point(x,y);
32:             }
33:     }
34:     pause("EX8_2");
35: }
```

リスト 8-3　EX8_3.cpp

```
 1: //  球体にバンプマッピングする
 2:
 3: #include        "graph1.h"
 4: #include        "graph3.h"
 5: #include        "ray.h"
 6:
 7: //  マッピング座標を返す
 8: Point   mapping(Win2D& w, Vector& V, int
 9:         float   a=0,          e=asin(V.z);               // a:方位角 e:仰角
10:         int     wi=w.size_x,hi=w.size_y;
11:         if      (V.y>0)  a= acos(V.x/sqrt(V.x*V.x+V.y*V.y));
12:         else if(V.y<0)  a=-acos(V.x/sqrt(V.x*V.x+V.y*V.y));
13:         return  Point( (int)(dir*a*wi/PI/2)%wi, e*hi/PI );
14: }
15:
16: main()
17: {
18:     Ball    Ba= Ball( Point(0,0,0),100);                  // 球体の記述
19:
20:     Win2D   bum("","..¥¥pic¥¥bump.bmp");                  // バンプ画像用ウィンドウ
21:
22:     Win3D   w("EX8_3", 0.25*CYAN);
23:     Point   V0=Point(500,200,200);                        // 視点の記述
24:     Point   L0=Point(500,-200,200);                       // 照明光の記述
25:     w.setview(V0);
26:
27:     for(int x=0; x<w.size_x; x++)
28:         for(int y=0; y<w.size_y; y++){
29:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);// 視線ベクトルの計算
30:             Ray     VR=Ray(V0,V);                         // 視線の記述
31:
32:             Point   Ps,Pr;                                // 交点座標
33:             Vector  N;                                    // 法線ベクトル
34:             if(Ba.hit(VR,Ps,N)<INFINITY){                 // 視線が球体と交わるかの判定
35:                 Vector  L=unit(Ps-L0);                    // 照明光の方向ベクトル
36:
37:                 Point   M= mapping(bum,N);                // マッピング画像の座標
38:
39:                 float   l00=5*(bum.pixel(M)).r;
40:                 float   l10=5*(bum.pixel(M+Point(1,0,0))).r;
41:                 float   l01=5*(bum.pixel(M+Point(0,1,0))).r;
42:
43:                 Vector  DX=Vector(1,0,l10-l00);
44:                 Vector  DY=Vector(0,1,l01-l00);
45:                 Vector  F =rot_z(90)*unit(DX%DY);
```

```
46:                     N=rotate(Point(0,0,1),F)*N;
47:
48:               w.color( shading(V,L,N,RED) );
49:               w.point(x,y);
50:          }
51:     }
52:     pause("EX8_3");
53: }
```

リスト 8-4　EX8_4.cpp

```
 1: //  球体に環境マッピング(リフレクションマッピング)する
 2: #include        "graph1.h"
 3: #include        "graph3.h"
 4: #include        "ray.h"
 5:
 6: //  マッピング座標を返す
 7: Point   mapping(Win2D& w, Vector& V, int
 8:         float   a=0,         e=asin(V.z);               //  a:方位角 e:仰角
 9:         int     wi=w.size_x,hi=w.size_y;
10:         if      (V.y>0)  a= acos(V.x/sqrt(V.x*V.x+V.y*V.y));
11:         else if(V.y<0)  a=-acos(V.x/sqrt(V.x*V.x+V.y*V.y));
12:         return  Point( (int)(dir*a*wi/PI/2)%wi, e*hi/PI );
13: }
14:
15: main()
16: {
17:     Win2D   tex("","..¥¥pic¥¥snow.bmp");                //  マッピング画像用ウィンドウ
18:     Ball    Ba= Ball( Point(0,0,0),70);                //  球体の記述
19:
20:     Win3D   w("EX8_4");
21:     Point   V0=200*unit(60,5);                         //  視点の記述
22:     Point   L0=Point(0,400,1000);                      //  照明光の記述
23:     w.setview(V0);
24:
25:     for(int x=0; x<w.size_x; x++)
26:         for(int y=0; y<w.size_y; y++){
27:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);// 視線ベクトルの計算
28:             Ray     VR=Ray(V0,V);                      //  視線の記述
29:
30:             Point   Ps,Pr;                             //  交点座標
31:             Vector  N;                                 //  法線ベクトル
32:             Color   Cs,Cr;
33:             if(Ba.hit(VR,Ps,N)<INFINITY){              //  視線が球体と交わるかの判定
34:                 Vector  L=unit(Ps-L0);                 //  照明光の方向ベクトル
35:                         Cs=shading(V,L,N,BLACK,1);
36:                 Vector  Vr=V-2*(V*N)*N;                //  反射視線の記述
37:
38:                 Point   C= mapping(tex,Vr,-1);         //  マッピング画像の座標
39:                         Cr= tex.pixel(C);              //  環境画像の色
40:             }
41:             else{
42:                 Point   M= mapping(tex,V,-1);          //  マッピング画像の座標
43:                         Cs=tex.pixel(M);               //  環境画像の色
44:             }
45:             w.color(Cs+Cr);
46:             w.point(x,y);
47:     }
48:     pause("EX8_4");
49: }
```

リスト 8-5　EX8_5.cpp

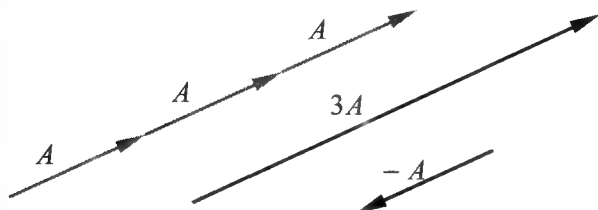
```
1: //  球体に環境マッピング(リフラクションマッピング)する
```

```
 2: #include       "graph1.h"
 3: #include       "graph3.h"
 4: #include       "ray.h"
 5:
 6: //  マッピング座標を返す
 7: Point   mapping(Win2D& w, Vector& V, int
 8:         float   a=0,        e=asin(V.z);              //  a:方位角 e:仰角
 9:         int     wi=w.size_x,hi=w.size_y;
10:         if      (V.y>0)   a= acos(V.x/sqrt(V.x*V.x+V.y*V.y));
11:         else if(V.y<0)   a=-acos(V.x/sqrt(V.x*V.x+V.y*V.y));
12:         return  Point( (int)(dir*a*wi/PI/2)%wi, e*hi/PI );
13: }
14:
15: main()
16: {
17:     Win2D   tex("","..¥¥pic¥¥snow.bmp");                    //  マッピング画像用ウィンドウ
18:     Ball    Ba= Ball( Point(0,0,0),70);                    //  球体の記述
19:     float   n=1.5;
20:
21:     Win3D   w("EX8_5");
22:     Point   V0=200*unit(60,5);                              //  視点の記述
23:     Point   L0=Point(0,400,1000);                           //  照明光の記述
24:     w.setview(V0);
25:
26:     for(int x=0; x<w.size_x; x++)
27:         for(int y=0; y<w.size_y; y++){
28:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);// 視線ベクトルの計算
29:             Ray     VR=Ray(V0,V);                           //  視線の記述
30:
31:             Point   Ps,Pr;                                  //  交点座標
32:             Color   Cs,Cr,Ct;
33:             Vector  N;                                      //  球体表面の放線ベクトル
34:             if(Ba.hit(VR,Ps,N)<INFINITY){                  //  視線が球体と交わるかの判定
35:                 Vector  L=unit(Ps-L0);                      //  照明光の方向ベクトル
36:                         Cs=shading(V,L,N,BLACK,1);
37:
38:                 float   c=-V*N, g=sqrt(n*n+c*c-1);
39:                 Vector  V2=(V+(c-g)*N)/n;
40:                 Point   P2=Ps-2*Ba.r*(V2*N)*V2;
41:                 Vector  N2=unit(Ba.o-P2);
42:                 float   c2=-V2*N2;
43:                 float   g2=sqrt(1/(n*n)+c2*c2-1);
44:                 Vector  V3=(V2+(c2-g2)*N2)*n;
45:                 Point   C= mapping(tex,V3,-1);              //  マッピング画像の座標
46:                         Ct=tex.pixel(C);                    //  環境画像の色
47:
48:                 Vector  Vr=V-2*(V*N)*N;                     //  反射視線の記述
49:                 Point   M= mapping(tex,Vr,-1);              //  マッピング画像の座標
50:                         Cr=tex.pixel(M);                    //  環境画像の色
51:             }
52:             else{
53:                 Point   M= mapping(tex,V,-1);               //  マッピング画像の座標
54:                         Cs=tex.pixel(M);                    //  環境画像の色
55:             }
56:             w.color(Cs+0.7*Ct+0.3*Cr);
57:             w.point(x,y);
58:     }
59:     pause("EX8_5");
60: }
```

# 付録１．ベクトルの演算

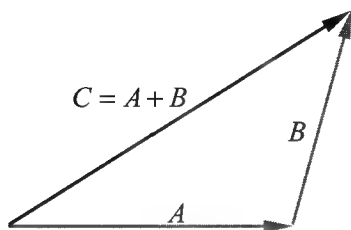
ベクトルは，方向を持っている量を表しています．そのため，ベクトル$A$は，$X,Y,Z$軸方向の各成分$A_x,A_y,A_z$で記述することができます．ベクトルに対して，一般に使用している長さや大きさなどの量をスカラーといいます．プログラムでベクトルを記述するために，ベクトルクラス(Pointクラスと同じ)を用意しています．また，以下に示すような演算ができるように，演算子を用意しています（付録2.3参照）．

## １．１　かけ算（スカラー倍）

ベクトル$A$を３倍した場合，図のようにベクトル$A$を３つそれぞれ終点と始点を重ねたもの（加算と同じ）となります．ベクトル$A$を，方向は変えずに３倍に伸ばしたものとなります．かけ算の記号は，プログラム例では"*"と記述しています．ベクトル$A$を$n$倍したのもをベクトル$B$とすると，各成分は下式のようになります．$n$が負の場合には，方向が逆になります．

$$B_x = nA_x$$
$$B_y = nA_y$$
$$B_z = nA_z$$

ベクトルのかけ算（スカラー倍）

## １．２　加算，減算

ベクトル$A$とベクトル$B$を加算したベクトル$C$は，図のようになります．ベクトル$A$の終点とベクトル$B$の始点を重ね，ベクトル$A$の始点とベクトル$B$の終点を結んだものがベクトル$C$となります．ベクトル$A$からベクトル$B$を減算する場合は，ベクトル$A$にベクトル$-B$を加算するものと考えます．加算の記号"＋"，減算の記号"－"は，プログラム例でも同様に記述しています．各成分に分けて書くと下式のようになります．

$$C_x = A_x + B_x$$
$$C_y = A_y + B_y$$
$$C_z = A_z + B_z$$

ベクトルの加算

ベクトルの減算

また，次のような法則が成り立ちます．

$$A + B = B + A$$
$$(A + B) + C = A + (B + C)$$

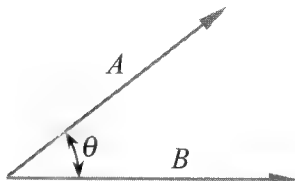
## １．３　内積（スカラー積）

図のようにベクトル $A$ とベクトル $B$ のなす角が $\theta$ の場合，ベクトルの内積は以下のような値（スカラー量）になります．

$$A \cdot B = |A| \cdot |B| \cdot \cos\theta$$

$$|A| = \sqrt{{A_x}^2 + {A_y}^2 + {A_z}^2}$$

$$|B| = \sqrt{{B_x}^2 + {B_y}^2 + {B_z}^2}$$

内積の記号"・"を，プログラム例では"*"と記述しています．かけ算（スカラー倍）と同じ記号ですが，ベクトルとベクトルの演算の場合には内積が行われ，実数とベクトルの演算の場合にはかけ算（スカラー倍）が行われます．各成分に分けて書くと下式のようになります．

ベクトルの内積

$$A \cdot B = A_x \cdot B_x + A_y \cdot B_y + A_z \cdot B_z$$

また，次のような法則が成り立ちます．

$$A \cdot B = B \cdot A$$
$$(mA) \cdot B = A \cdot (mB) = m(A \cdot B)$$
$$(-A) \cdot B = A \cdot (-B) = -A \cdot B$$
$$A \cdot (B + C) = A \cdot B + A \cdot C$$

$$A \cdot A = |A|^2$$

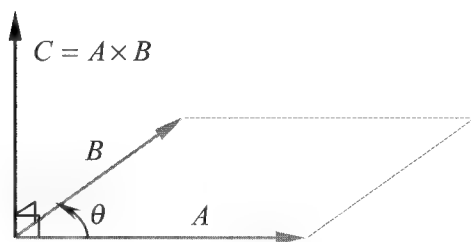
$|A| = 1, |B| = 1$ の場合　$A \cdot B = \cos\theta$

## 1.4　外積（ベクトル積）

ベクトル $A$ とベクトル $B$ の外積ベクトル $C$ は，図のようになります．ベクトル $A$ とベクトル $B$ を含む平面に垂直で，右ねじをベクトル $A$ からベクトル $B$ に回転させた場合に進む方向のベクトルとなります．そのベクトルの絶対値は，下式のように，ベクトル $A$，$B$ を辺とする平行四辺形の面積になります．

$$|A \times B| = |A| \cdot |B| \cdot \sin\theta \qquad (0 \leq \theta < 180)$$

外積の記号"×"を，プログラム例では"%"と記述しています．

各成分に分けて書くと下式のようになります．

$$C_x = A_y \cdot B_z - A_z \cdot B_y$$
$$C_y = A_z \cdot B_x - A_x \cdot B_z$$
$$C_z = A_x \cdot B_y - A_y \cdot B_x$$

ベクトルの外積

また，次のような法則が成り立ちます．

$$A \times B = -B \times A$$
$$(mA) \times B = A \times (mB) = m(A \times B)$$
$$A \times (B + C) = (A \times B) + (A \times C)$$

$A \times A = O$（各成分が０）

$|A| = 1, |B| = 1$ の場合　$|A \times B| = \sin\theta$

# 付録2. クラス，関数一覧

## 2.1　クラスの一覧

| クラス名 | 内容 | 記述されているファイル |
|---|---|---|
| Attri | 光学的性質 | attri.h |
| Ball | 球体 | ray.h |
| Color | 色 | win.h |
| Csgm | ２次曲面のCSGモデル | csgm.h |
| Lay | 光線，視線 | ray.h |
| Line | 複数線分 | line.h |
| Point | 点，座標 | point.h |
| Polygon | 多角形 | polygon.h |
| Scene | 場面を記録する | scene.h |
| Surface | サーフェイスモデル | surface.h |
| TMatrix | 座標変換行列 | tmatrix.h |
| Vector | ベクトル（Pointと同じ） | point.h |
| Wall | 壁，床（多角形） | ray.h |
| Win | ウィンドウ（スクリーン座標） | win.h |
| Win2D | ウィンドウ（２次元標準座標） | graph1.h |
| Win3D | ウィンドウ（３次元標準座標） | graph2.h, graph3.h |

## 2.2　関数一覧（各クラスのメンバ関数以外）

| 関数のプロトタイプ | | 記述されているファイル | |
|---|---|---|---|
| Csgm | ball(Point& s); | csgm.h | 楕円面 |
| Csgm | ball(float r); | sgm.h | 球体 |
| Csgm | board_xy(float b=10); | csgm.h | XY平面の板 |
| Line | circle_xy(int n, float b=0, float e=360); | line.h | XY平面上の円弧 |
| Line | circle_yz(int n, float b=0, float e=360); | line.h | YZ平面上の円弧 |
| Line | circle_xz(int n, float b=0, float e=360); | line.h | XZ平面上の円弧 |
| Csgm | cylinder(Point& s); | csgm.h | 楕円柱面 |
| float | len(Line& a); | line.h | 大きさ(長さ)を返す |
| float | len(Point& a); | point.h | 大きさ(長さ)を返す |
| float | max(float a, float b); | graph3.h | 大きい方を返す |
| float | min(float a, float b); | graph3.h | 小さい方を返す |
| TMatrix | move(Point& a); | tmatrix.h | 平行移動 |
| TMatrix | move(float x, float y, float z); | tmatrix.h | 平行移動 |
| Csgm | plane(Point& s, float b=0); | csgm.h | 平面 |
| Csgm | plane_xy(); | csgm.h | XY平面 |
| Csgm | plane_yz(); | csgm.h | YZ平面 |
| Csgm | plane_xz(); | csgm.h | XZ平面 |
| void | pause(char *txt="ＯＫで終了します！"); | win.h | 一時停止(ボタン入力待ち関数) |
| TMatrix | perspect(float d); | tmatrix.h | 透視変換（遠近感） |
| Surface | revolve_z(Line& a, int n, float b=0, float e=360); | surface.h | aを回転したモデル |
| TMatrix | rotate(Point& n); | tmatrix.h | ベクトル方向に回転 |
| TMatrix | rotate(Point& a, float r); | tmatrix.h | 回転変換 |
| TMatrix | rotate(Point& a, Point& b); | tmatrix.h | 回転変換 |

| | | | |
|---|---|---|---|
| TMatrix | rotate(Point& a, float sn, float cs); | tmatrix.h | 回転変換 |
| TMatrix | rot_x(float r); | tmatrix.h | X軸回転 |
| TMatrix | rot_x(float sn, float cs); | tmatrix.h | X軸回転 |
| TMatrix | rot_y(float r); | tmatrix.h | Y軸回転 |
| TMatrix | rot_y(float sn, float cs); | tmatrix.h | Y軸回転 |
| Tmatrix | rot_z(float r); | tmatrix.h | Z軸回転 |
| TMatrix | rot_z(float sn, float cs); | tmatrix.h | Z軸回転 |
| TMatrix | rotate(float r); | tmatrix.h | Z軸回転 |
| Tmatrix | scale(Point& a); | tmatrix.h | スケール変換 |
| TMatrix | scale(float x, float y, float z); | tmatrix.h | スケール変換 |
| Color | shading(Vector& V, Vector& L, Vector& N, Color& c, float s=1); | ray.h | 描画色の計算 |
| Color | shading(Vector& V, Vector& L, Vector& N, Attri& A, float s=1); | attri.h | 描画色の計算 |
| Surface | sweep_xz(Line& a, Line& b); | surface.h | aをbに沿って押し出したモデル |
| Point | unit(Point& a); | point.h | 大きさを１にする |
| Point | unit(float a, float e); | point.h | 方位角a, 仰角eの単位ベクトル |

## 2.3 演算子一覧 (point.h, line.h, tmatrix.h, polygon.h, surface.h, csgm.h)

| 演算子記号 | 左辺のクラス(a) | 右辺のクラス(b) | 演算結果のクラス | 処理内容 |
|---|---|---|---|---|
| + | Point | Point | Point | 各座標値を加算する |
| - | Point | Point | Point | 各座標値を減算する |
| * | float | Point | Point | bの各座標値をa倍する |
| * | Point | float | Point | aの各座標値をb倍する |
| * | Point | Point | float | 内積 |
| / | Point | float | Point | aの各座標値をbで割る |
| % | Point | Point | Point | 外積 |
| == | Point | Point | int | aとbの各座標値が等しいかいなか |
| * | float | Line | Line | bの各座標値をa倍する |
| * | TMatrix | Point | Point | bの座標変換を行う |
| * | Tmatrix | Line | Line | bの座標変換を行う |
| * | TMatrix | Polygon | Polygon | bの座標変換を行う |
| * | TMatrix | TMatrix | TMatrix | 行列の積をとる |
| * | TMatrix | Polygon | Polygon | bの座標変換を行う |
| * | TMatrix | Surface | Surface | bの座標変換を行う |
| - | Csgm | | Csgm | 立体の内部と外部を逆にする (単項演算子) |
| + | Csgm | Csgm | Csgm | 立体ａと立体ｂの和 |
| - | Csgm | Csgm | Csgm | 立体ａに含まれ，立体ｂに含まれない部分の立体 ａ＊(－ｂ)と同じ |
| * | Csgm | Csgm | Csgm | 立体ａと立体ｂの積 |
| * | TMatrix | Csgm | Csgm | 立体ａの座標変換(移動, 回転, 拡大など) |

# 付録3．クラスの拡張

クラスなどを拡張したヘッダファイルを，CG_×××¥extまたはCG_×××¥incに用意しています．付録4.2～4.4のプログラム例では，本項で説明する拡張を利用しています．

## 3．1　printf, scanf関数の使用

ヘッダファイルwin2.hをwin.hの代わりにインクルードすると，printf,scanf関数が使用できるようになります．printf関数のパフォーマンスが低いので，大量に出力すると動作が遅くなります．また，scanf関数は，入力できる変数が最大11という制約があります．printf,scanf関数を実行するとウィンドウが開き，入出力ができるようになります．このウィンドウはスクロールができ，表示結果をテキスト形式でファイルに保存することができます．

## 3．2　描画方法の拡張

Surfaceクラス（ポリゴンモデル），Csgmクラス（CSGモデル），Ballクラス（球体），Wallクラス（壁）のデータを描画できるように，Win3D（ヘッダファイルgraph4.h）を拡張しました．

Surfaceクラスのデータは，ワイヤーフレーム，フラットシェーディング，グーローシェーディング，フォンシェーディングによる描画，およびシャドーマッピングによる影付けができるようにしました．
また，Suerfaceクラスのデータをレイトレーシング法を用いて，多面体および曲面（*B*スプライン）として描画できるようにしました（内部的に自由曲面用のFreeクラスに変換しており，バウンディングボリュームを使用して描画の高速化を図っています）．

Csgmクラス，Ballクラス，Wallクラスのデータは，レイトレーシング法で描画することができます．さらに，レイトレーシング法では環境マッピングができるようにし，またWallクラスのデータにはテクスチャマッピングができるようにしました．

物体（形状，光学的性質，配置）を記録するためのSceneクラス（scene2.h）を用意しており，Win3Dクラスのrender関数で描画を行います．描画を行う手順は，以下の通りです．

① 物体(形状，光学的性質，配置)を記録するSceneクラス（scene2.h）で環境マッピング画像などを指定する（コンストラクタで指定する）．

```
Scene::Scene(Color& b=BLACK, int m=1);
Scene::Scene(char* fn, int m=1, float off=0);
```

| | |
|---|---|
| fn: | 背景画像のファイル名 |
| c: | 背景色 |
| m: | 0:背景描画なし 1:背景描画あり |
| off: | 背景画像の位置指定（0～1） |

② regist関数によって物体を登録します．

```
Void    Scene::regist(Wall& a, Attri& c);
void    Scene::regist(Ball& a, Attri& c);
void    Scene::regist(Csgm& a, Attri& c, Ball& bv=Ball(Point(0,0,0), -1));
void    Scene::regist(Free& a, Attri& c, int m=SMOOTH);
void    Scene::regist(Surface& as, Surface& an, Attri& c);
```

```
void       Scene::regist(Surface& as, Attri& c, int m=SMOOTH);
           as: 形状データ
           an: 法線ベクトル
           c:  光学的性質
           bv: バウンディングボリューム(Ballクラス)
           m:  形状データの属性（下記を指定）
               #define     SMOOTH      0x00滑らかな曲面
               #define     FLAT        0x02多面体
```

③ render関数によって描画を行います．

```
Void       Win3D::render( Scene& sc, int md=POLYGON, int a=PHONG);
  ・ポリゴンモデルの場合の指定
   md:下記を指定
               #define     POLYGON     0x100   ポリゴン
               #define     POLY        0x100   ポリゴン
                           (md=POLYGONでは, CsgmおよびBallクラスは描画できません)
   a:下記の描画方法を指定
               #define     WIRE        0x01ワイヤーフレーム
               #define     FLAT        0x02フラットシェーディング
               #define     GURO        0x03グーローシェーディング
               #define     PHONG       0x04フォンシェーディング
               #define     SHADOW      0x00影付けあり(上項目と"|"で組み合わせる)
               #define     NOSHADOW    0x10影付けなし(上項目と"|"で組み合わせる)
               #define     NOS         0x10影付けなし(上項目と"|"で組み合わせる)
   ・レイトレーシングの場合の指定
    md:下記を指定
               #define     RAYTRACING  0x200   レイトレーシング
               #define     RAY         0x200   レイトレーシング
    a:レイトレーシングでの反射，屈折の制限回数を指定
```

## 3.3　連結演算子および関数の追加

ヘッダファイルextop.hに，以下の演算子および関数を記述しています．LineクラスとPointクラス間の連結演算子("|","||")によって，Lineクラスを生成することができます．また，SurfaceクラスとPolygonクラス間の連結演算子("||")によって，Surfaceクラスを生成することができます．（プログラム例AP2_2で，具体例を説明しています）

| 演算子記号 | 左辺のｸﾗｽ(a) | 右辺のｸﾗｽ(b) | 演算結果のｸﾗｽ | 処理内容 |
|---|---|---|---|---|
| \| | Line | Line | Line | aとbを連結する　(bを移動しaの終点とbの始点を重ねる) |
| \| | Line | Point | Line | aにbを追加する　(bはaの終点からの相対位置を表す) |
| \| | Point | Point | Line | aとbの線分を作る　(bはaからの相対位置を表す) |
| \|\| | Line | Line | Line | aとbを連結する　(bは絶対位置を表す) |
| \|\| | Line | Point | Line | aにbを追加する　(bは絶対位置を表す) |
| \|\| | Point | Point | Line | aとbの線分を作る　(bは絶対位置を表す) |
| \|\| | Surface | Surface | Surface | aとbの結合する |
| \|\| | Surface | Polygon | Surface | aとbの結合する |
| \|\| | Polygon | Surface | Surface | aとbの結合する |

# 付録４．プログラム例

ここで説明するプログラム例は，CG_×××¥extまたはCG_×××¥prjに格納されており，付録４での拡張を利用しています．

## ４．１　２次元のプログラム例(AP1_*.cpp)

Win2Dを使用したプログラム例(AP1_1からAP1_5)を用意しています．プロジェクトAP1で実行することができ，内容は以下の通りです．

AP1_1　花形図形

以下のように，半径を変化させながら$(x, y)$の軌跡を描きます．

$$r = 100 \cdot \sin(\frac{ij}{k}\pi),\quad j = 8, k = 9, i = 0 \sim k$$

$$x = r \cdot \cos(\frac{2\pi i}{360}),\quad y = r \cdot \sin(\frac{2\pi i}{360})$$

AP1_2　再帰図形（円の中に円）

円の中に，３個（$m$個）の円を描きます．その各円の中に，さらに３個（$m$個）の円を描きます．これを繰り返します．

AP1_3 コッホ曲線

正６角形（正多角形）の各辺を，図形"＿／＼＿"で置き換えます．置き換えた図形の各辺を，さらに同様に置き換えます．これを繰り返します．

AP1_4　自己平方フラクタル

複素数平面$(x, y_i)$で　$a = -0.3 - 0.6i, z = x + y_i$の場合，　$z = z * z + a$の計算を繰り返し，$|z| > 2$となったときの繰り返し回数で色を付けます．

Visual C++では，complexクラス(複素数クラス)で演算ができないので，compクラスを定義して行っています．

AP1_5　マンデルブロー集合

複素数平面$(x, y_i)$で　$a = x + y_i, z = 0$の場合，　$z = z * z + a$の計算を繰り返し，$|z| > 2$となったときの繰り返し回数で色を付けます．

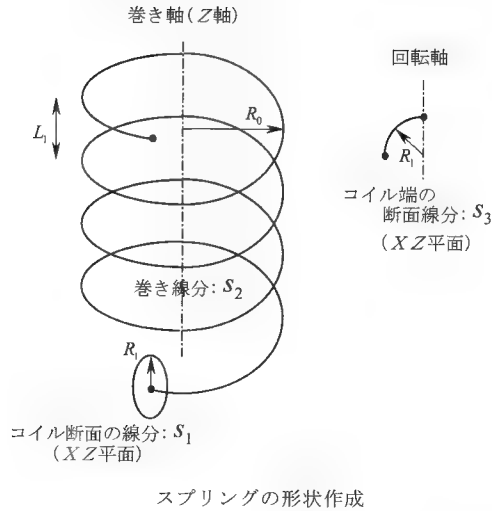
Visual C++では，complexクラス(複素数クラス)で演算ができないので，compクラスを定義して行っています．

## ４．２　ポリゴンモデルのプログラム例(AP2_*.cpp)

ポリゴンモデルを使用したプログラム例(AP2_1からAP2_3)を用意しています．プロジェクトAP2で実行することができ，内容は以下の通りです．

AP2_1　スプリング

らせん円筒と半球を部品として作り，それを組み合わせてスプリングを描画しています．影付けなしおよび影付けした描画です．形状の作成手順を以下に示します．

スプリングの形状作成

① らせん円筒の作成

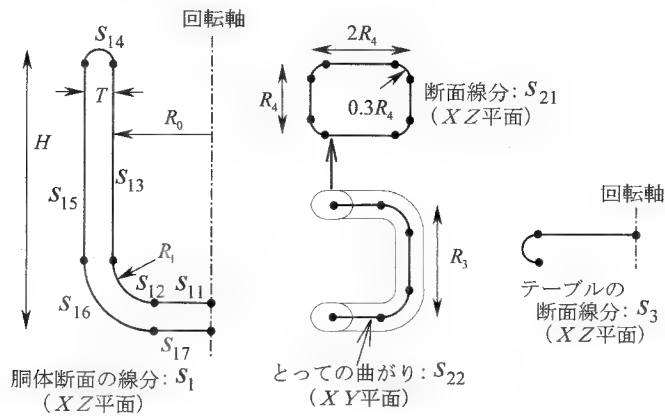
・図のように，断面線分 $s_1$ を巻き線分 $s_2$ に沿ったらせん円筒形状 $p_1$ を作成(sweep_xz関数を使用)する．

② 半球（両端）の作成

・図のように，断面線分 $s_3$ をを回転して半球形状 $p_2, p_3$ を作成(revolve_z関数を使用)する．

AP2_2　コーヒーカップ

ワイヤーフレーム，フラットシェーディング，グーローシェーディング，フォンシェーディングにより描画しています．コーヒーカップの形状の作成手順を以下に示します．

コーヒーカップの形状作成

① 胴体の作成

・図のように，線分 $s_{11} \sim s_{17}$ のを連結（連結する演算子"|"を使用)して断面 $s_1$ を作る($XZ$平面上).

・断面 $s_1$ を回転して胴体形状 $p_1$ を作成（revolve_z関数を使用）する．

② とっての作成

・線分を連結して，とっての断面 $s_{21}$ を作成する（$XZ$平面上）．

・線分を連結して，とっての曲がり $s_{22}$ を作成する（$XY$平面上）．

・断面 $s_{21}$ を曲がり $s_{22}$ に沿ったとっての形状 $p_2$ を作成（sweep_xz関数を使用）する．

③ テーブルの作成

・線分を連結して，テーブルの断面 $s_3$ を作成する（$XZ$平面上）．

・断面 $s_3$ を回転してテーブルの形状 $p_3$ を作成（revolve_z関数を使用）する．

AP2_3　正２０面体

正三角形による正２０面体($N$=5)を描画します．また，各正三角形を４つの三角形($M$=1)で置き換えて，三角形で球体を描きます．プログラム中の変数$N$と$M$を変更することによって，形状が変わります．

$N$は正三角形で構成されるベースとなる形状を表します．ベースとなる形状を構成する各正三角形を４の$M$乗の三角形で置き換えます．

| | | | |
|---|---|---|---|
| $N=5$ | 正２０面体 | $M=0$ | 置き換えない |
| $N=4$ | 正８面体 | $M=1$ | ４つの三角形で置き換える |
| $N=3$ | 正４面体 | $M=2$ | １６の三角形で置き換える |

## 4.3　ＣＳＧモデルのプログラム例（AP3_*.cpp）

ＣＳＧモデルをレイトレーシングで描画するプログラム例（AP3_1からAP3_5）を用意しています．プロジェクトAP3で実行することができ，内容は以下の通りです．

AP3_1　２次曲面

以下に示す９つの２次曲面を描画しています．

各２次曲面は，下記のように表現することができます．

２次曲面とその方程式

| | $f(x,y,z)$ | $S_{2x},S_{2y},S_{2z}$ | $S_{1x},S_{1y},S_{1z}$ | $S_0$ |
|---|---|---|---|---|
| ① 楕円面 | $\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}-1$ | $\frac{1}{a^2},\frac{1}{b^2},\frac{1}{c^2}$ | 0,0,0 | −1 |
| ② 一葉双曲面 | $\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}-1$ | $\frac{1}{a^2},\frac{1}{b^2},-\frac{1}{c^2}$ | 0,0,0 | −1 |
| ③ 二葉双曲面 | $-\frac{x^2}{a^2}-\frac{y^2}{b^2}+\frac{z^2}{c^2}-1$ | $-\frac{1}{a^2},-\frac{1}{b^2},\frac{1}{c^2}$ | 0,0,0 | −1 |
| ④ 楕円錐面 | $\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z^2}{c^2}$ | $\frac{1}{a^2},\frac{1}{b^2},-\frac{1}{c^2}$ | 0,0,0 | 0 |
| ⑤ 双曲放物面 | $\frac{x^2}{a^2}+\frac{y^2}{b^2}-\frac{z}{c}$ | $\frac{1}{a^2},\frac{1}{b^2},0$ | $0,0,-\frac{1}{c}$ | 0 |
| ⑥ 楕円放物面 | $\frac{x^2}{a^2}-\frac{y^2}{b^2}-\frac{z}{c}$ | $\frac{1}{a^2},-\frac{1}{b^2},0$ | $0,0,-\frac{1}{c}$ | 0 |
| ⑦ 放物柱面 | $\frac{x^2}{a^2}+\frac{y}{b}$ | $\frac{1}{a^2},0,0$ | $0,\frac{1}{b},0$ | 0 |
| ⑧ 楕円柱面 | $\frac{x^2}{a^2}+\frac{y^2}{b^2}-1$ | $\frac{1}{a^2},\frac{1}{b^2},0$ | 0,0,0 | −1 |
| ⑨ 双曲柱面 | $\frac{x^2}{a^2}-\frac{y^2}{b^2}-1$ | $\frac{1}{a^2},-\frac{1}{b^2},0$ | 0,0,0 | −1 |

レイトレーシング法での視線（視点$V_0$，方向ベクトル$V$）と２次曲面の交点をＰとすると，以下のような関係になります（Ｐは点の座標とともに，原点からのベクトルも表すものとします）．

$$V_0+t\cdot V=\mathrm{P}$$

また，各２次曲面を描画するために方程式　$f(x,y,z)=0$　について考え，各方程式の係数を

$S_{2x},S_{2y},S_{2z}$，$S_{1x},S_{1y},S_{1z}$，$S_0$とすると，下式のように一般化して表現することができます．

（$S_2,S_1$はベクトルとして表現します）

$$(\mathrm{scale}(S_2)\cdot P)\cdot P+S_1\cdot P+S_0=0$$

すると，下式のような２次方程式となります．

$$\{(\mathrm{scale}(S_2)\cdot V)\cdot V\}t^2 + 2\{(\mathrm{scale}(S_2)\cdot V)\cdot V_0 + S_1\cdot V/2\}t + (\mathrm{scale}(S_2)\cdot V_0)\cdot V_0 + S_1\cdot V_0 + S_0 = 0$$

判別式 $d$ と $t$ は以下のようになります．

$$d = \{(\mathrm{scale}(S_2)\cdot V)\cdot V_0 + S_1\cdot V\}^2 - \{(\mathrm{scale}(S_2)\cdot V)\cdot V\}\{(\mathrm{scale}(S_2)\cdot V_0)\cdot V_0 + S_1\cdot V_0 + S_0\}$$

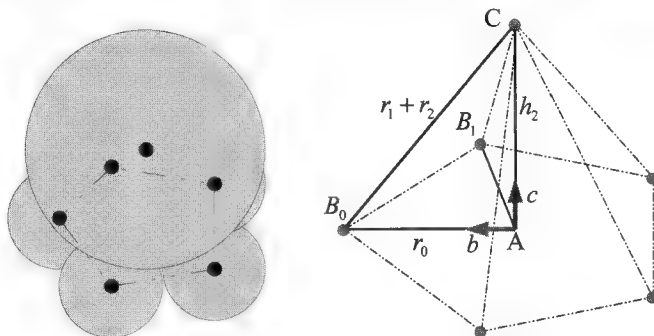
$$t = -\frac{(\mathrm{scale}(S_2)\cdot V)\cdot V_0 + 0.5S_1\cdot V}{(\mathrm{scale}(S_2)\cdot V)\cdot V} - \frac{\sqrt{d}}{|(\mathrm{scale}(S_2)\cdot V)\cdot V|}$$

法線ベクトル $N$ は，以下のようになります．

$$N = \mathrm{unit}(2\cdot\mathrm{scale}(S_2)\cdot P + S_1)$$

プログラムでは，常に表側が見えるように法線ベクトル $N$ を選択しています．

## AP3_2　積み重ねた球体

積み重ねた球体を描画します．図のように，球体の上に球体を重ねる場合の位置関係を，以下に示します．

以下のパラメータから，球体 $B_i$ と $C$ の中心座標を求めることができます．

$A$　：球体を配置する基準位置

$b$　：下の球体 $B_0$ を配置する方向（$X$あるは $Y$軸方向）

$r_1$　：下の球体 $B_i$ の半径

$n$　：下の球体の個数

$c$　：上の球体 $C$ を配置する方向（$b$ と $c$ は直交している必要がある：$Z$軸方向）

$r_2$　：上の球体 $C$ の半径

$B_i$　：下の球体の中心位置（座標）

$$B_i = A + \mathrm{rot_z}\left(\frac{360}{n}i\right)\cdot(r_0\cdot b)$$

$$r_0 = \frac{r_1}{\sin(\pi/n)}$$

$C$：上の球体の中心位置（座標）

$$C = A + h\cdot c$$

$$h = \sqrt{(r_1 + r_2)^2 - r_0^2}$$

AP3_3　透明な球体

AP3_2を変更して，上部の球体を透明なものにしました．

AP3_4　レンズ

凸レンズおよび凹レンズを通して景色を見ています．

AP3_5　テーブルと灰皿

ガラステーブルの上に灰皿があります．また，壁と床には，テクスチャーマピングを行っています．

## 4.4　自由曲面（$B$スプライン）のプログラム例(AP4_*.cpp)

自由曲面（$B$スプライン）をレイトレーシングで描画するプログラム例(AP4_1からAP4_3)を用意しています．プロジェクトAP4で実行することができ，内容は以下の通りです．

AP4_1～AP4_3はそれぞれAP2_1～AP2_3と同じ形状を描画します．環境マッピングを行い，金属のように反射する場合と，ガラスのように透明な場合について描画します．描画時間が，数分から数十分かかります（233MHzのPentiumⅡでVisual C++ 4.0(Developer Studio)を使用した場合，AP2_1は約３分，P2_2は約４分，AP2_3は約３分かかります）．

リスト　AP1_1.cpp

```
 1: //      簡単な図を画く（座標の演算を含む）
 2: //      リサジュー図形
 3:
 4: #include    "graph1.h"
 5:
 6: main()
 7: {
 8:     Win2D   w("AP1_1");                                 //  ウィンドウ生成
 9:     w.axis();                                           //  座標軸の表示
10:
11:     int j=9;
12:     int k=8;
13:
14:     w.color( RED );
15:     w.move ( Point(0,0) );
16:     for(int i=0; i<360*k; i+=2){
17:         float   r=100*sin(i*PIR*j/k);
18:         w.line ( r*unit(i) );
19:     }
20:     pause("AP1_1");
21: }
22:
```

リスト　AP1_2.cpp

```
 1: //      簡単な図を画く（座標の演算を含む）
 2: //      再帰図形
 3:
 4: #include    "graph1.h"
 5:
 6: //      円を描画する
 7: void    circle(Win2D& w, Point c, float r){
 8:         int n=(r+7)/2;
 9:         w.color( RED );
10:         w.move( c+r*Point( 1, 0) );
11:         for(int i=1; i<=n; i++) w.line( c+r*unit( 360.*i/n) ) ;
12: }
13:
14: //      再帰図形を描く（円の中にm個の円を描く
15: #define RR      0.45
16: void    recursive(Win2D& w, Point c, float r, int n)
17: {
18:     #define     NEST    4                               //  深さの段数
19:     if(n>NEST) return;
20:     circle( w, c, r );
21:
22:     int m=3;                                            //  円の中に描画する円の個数
23:     Point   p;
24:
25:     for( int i=0; i<m; i++) {
26:         p=c+r*(1.-RR)*unit( 360.*i/m );
27:         recursive( w, p, r*RR, n+1 );
28:     }
29: }
30:
31: main()
32: {
33:     Win2D   w("AP1_2");                                 //  ウィンドウ生成
34:     recursive( w, Point(0,0),150, 0);
35:     pause("AP1_2");
36: }
```

リスト　AP1_3.cpp

```
 1: //  コッホ曲線を画く
 2:
 3: #include    "graph1.h"
 4:
 5: #define NEST    4                                    //  深さの段数
 6: void    koch( Win2D& w, Point sp, Point ep, int n)
 7: {
 8:     if(n> NEST) return;
 9:     w.color( RED );
10:     if(n==NEST){    w.line(sp,ep);  return; }
11:
12:     //  ジェネレータの設定
13:     Point   p[5]={  Point(0,0),       Point(0.33,0),
14:                     Point(.5,.2886),Point(0.66,0),
15:                     Point(1,0)          };
16:
17:     //  ジェネレータの置き換え
18:     for( int i=0; i<4; i++)
19:         koch( w, rotate(ep-sp)*(len(ep-sp)*p[i])+sp,
20:              rotate(ep-sp)*(len(ep-sp)*p[i+1])+sp, n+1  );
21: }
22:
23: main()
24: {
25:     Win2D   w("AP1_3");
26:
27:     int     n=6;                                     //  基本図形（ｎ角形）
28:     float   r=150;                                   //  基本図形の半径
29:
30:     for(int i=0; i<n; i++)
31:         koch( w, r*unit(i*360./n),
32:              r*unit((i+1)*360./n) , 0);
33:
34:     pause("AP1_3");
35: }
```

リスト　AP1_4.cpp

```
 1: //  自己平方フラクタルを画く
 2: //  z=z*z+a を繰返して、集束する点（ｚ）を表示する
 3: //  ｚは複素数で、ｘ＋ｙｉ
 4:
 5: #include    "graph1.h"
 6:
 7: class   comp{                                        //  複素数を記述
 8:         public:
 9:         float   re,im;
10:         comp(float r, float i){ re=r;   im=i;   }
11: };
12:
13: comp    operator+(comp& a, comp& b)                  //  複素数の演算
14: {       return comp(a.re+b.re, a.im+b.im);}
15: comp    operator*(comp& a, comp& b)
16: {       return comp(a.re*b.re-a.im*b.im, a.re*b.im+a.im*b.re);}
17: float   abs(comp& a)
18: {       return sqrt(a.re*a.re+a.im*a.im);   }
19:
20: #define X_SIZE  400                                  //  ウィンドウサイズ
21: #define Y_SIZE  400
22: #define SCALE   100.                                 //  １のサイズ(dot)
23: #define R_LIM   (X_SIZE/SCALE/2)                     //  実数部の範囲
24: #define I_LIM   (Y_SIZE/SCALE/2)                     //  虚数部の範囲
25: #define N_LIM   1000                                 //  繰り返し回数
26:
```

```
27: main()
28: {
29:     Win2D   w("AP1_4",X_SIZE,Y_SIZE);                      // ウィンドウ生成
30:     w.axis ();                                             // 座標軸の表示
31:
32:     comp    a=comp(-0.3,-0.63);                            // 定数
33:
34:     w.color( RED );
35:     for(float imag=-I_LIM; imag<=I_LIM; imag+=1/SCALE)
36:         for(float real=-R_LIM; real<=R_LIM; real+=1/SCALE){
37:             comp    z=comp(real,imag);
38:             for(int i=1; i<N_LIM; i++){
39:                 z=z*z+a;
40:                 if(abs(z)>2)break;
41:             }
42:             if(i<N_LIM){
43:                 float   k=-log((float)i/N_LIM)/8;
44:                 if(k>1)k=1;
45:                 w.color((1-k)*YELLOW);
46:                 w.point( Point(real*SCALE,imag*SCALE) );
47:             }
48:         }
49:     w.axis();
50:     pause("AP1_4");
51: }
52:
```

リスト　AP1_5.cpp

```
 1: //  マンデルブロー集合を画く
 2: //  z=z*z+a を繰返して、集束する点（a）を表示する
 3: //  aは複素数で、X＋Yi　Zの初期値は０
 4:
 5: #include    "graph1.h"
 6:
 7: class   comp{                                               // 複素数を記述
 8:         public:
 9:         float   re,im;
10:         comp(float r, float i){ re=r;   im=i;   }
11: };
12:
13: comp    operator+(comp& a, comp& b)                         // 複素数の演算
14: {       return comp(a.re+b.re, a.im+b.im);}
15: comp    operator*(comp& a, comp& b)
16: {       return comp(a.re*b.re-a.im*b.im, a.re*b.im+a.im*b.re);}
17: float   abs(comp& a)
18: {       return sqrt(a.re*a.re+a.im*a.im);   }
19:
20: #define X_SIZE  400                                         // ウィンドウサイズ
21: #define Y_SIZE  400
22: #define SCALE   100.                                        // １のサイズ(dot)
23: #define R_LIM   (X_SIZE/SCALE/2)                            // 実数部の範囲
24: #define I_LIM   (Y_SIZE/SCALE/2)                            // 虚数部の範囲
25: #define N_LIM   1000                                        // 繰り返し回数
26:
27: main()
28: {
29:     Win2D   w("AP1_5",X_SIZE,Y_SIZE);                      // ウィンドウ生成
30:     w.axis();                                              // 座標軸の表示
31:
32:     for(float imag=-I_LIM; imag<=I_LIM; imag+=1/SCALE)
33:         for(float real=-R_LIM; real<=R_LIM; real+=1/SCALE){
34:             comp    a=comp(real,imag);
35:             comp    z=comp(0,0);
36:             for(int i=1; i<N_LIM; i++){
37:                 z=z*z+a;
```

```
38:                 if(abs(z)>2)break;
39:             }
40:             if(i<N_LIM){
41:                 float   k=-log((float)i/N_LIM)/8;
42:                 if(k>1)k=1;
43:                 w.color((1-k)*YELLOW);
44:                 w.point( Point(real*SCALE,imag*SCALE) );
45:             }
46:         }
47:     w.axis();
48:     pause("AP1_5");
49: }
50:
```

リスト AP2_1.cpp

```
 1: //  フォンシェーディング
 2: //  コイル・スプリングを描画する
 3:
 4: #include        "graph4.h"
 5:
 6: main()
 7: {
 8:     //  モデリング
 9:     //  コイルの作成
10:     #define     R0      50.                           //  コイルの巻き半径
11:     #define     R1      15.                           //  コイルの半径
12:     #define     RN      4                             //  巻き数
13:     #define     L1      60.                           //  一巻きの長さ
14:     #define     DEV0    24                            //  巻きの分割数
15:     #define     DEV1    12                            //  断面の分割数
16:
17:     Line    s1=R1*circle_xz( DEV1);                   //  コイルの断面
18:     Line    s2=Line(DEV0*RN+1);                       //  コイルを巻く軌跡
19:     for(int t=0; t<DEV0*RN+1; t++)
20:         s2.p[t]=rot_z(t*360./DEV0)*Point(R0,0,L1*t/DEV0);
21:     Surface p1=sweep_xz( s1, s2 );                    //  表面ポリゴン
22:
23:     //  端部品の作成
24:     Line    s3=circle_xz( DEV1/2, 0, 90);             //  円弧の生成
25:     Surface a1=rot_x(-90)*revolve_z(R1*s3,DEV1 );     //  表面ポリゴン
26:     Surface p2=move(startpoint(s2))*rot_xz(startvector(s2))*a1;
27:     Surface p3=move(endpoint(s2))*rot_xz(endvector(s2))*a1;
28:     TMatrix m=move(0,0,-130);
29:
30:     //  レンダリング
31:     Attri       At=Attri(RED, 0.7,0.7,0.3);
32:     Scene       sc;
33:     sc.regist( m*p1, At );
34:     sc.regist( m*p2, At );
35:     sc.regist( m*p3, At );
36:
37:     Win3D   w1( "AP2_1w1", WHITE*0.3+GREEN*0.1+BLUE*0.2);
38:     w1.render ( sc, POLYGON, PHONG|NOSHADOW);
39:     Win3D   w2( "AP2_1w2", WHITE*0.3+GREEN*0.1+BLUE*0.2);
40:     w2.render ( sc, POLYGON, PHONG);
41:
42:     pause("AP2_1");
43: }
44:
45:
```

リスト　AP2_2.cpp

```
 1: //  コーヒーカップを描画する
 2: #include        "graph4.h"
 3:
 4: main()
 5: {
 6:     //  モデリング
 7:     //  胴体の作成
 8:     #define     R0      60.                             //  カップの内径
 9:     #define     R1      25.                             //  カップ底の曲率
10:     #define     T       10.                             //  カップの厚さ
11:     #define     H       150.                            //  カップの高さ
12:     #define     R2      (T/2.)
13:     #define     L1      (H-T-R1-R2)
14:
15:     Line    s11=Point(0,0,T)|Point(R0-R1,0,0);          //  底        (内側)
16:     Line    s12=R1*circle_xz(6,180,90);                 //  底の丸み  (内側)
17:     Line    s13=Point(0,0,0)|Point(0,0,L1);             //  側面      (内側)
18:     Line    s14=R2*circle_xz(9,-90,90);                 //  上部
19:     Line    s15=Point(0,0,0)|Point(0,0,-L1);            //  側面      (外側)
20:     Line    s16=(R1+2*R2)*circle_xz(6,90,180);          //  底の丸み  (外側)
21:     Line    s17=Point(0,0,0)|Point(-R0+R1,0,0);         //  底        (外側)
22:     Line    s1=(s11|s12)|(s13|s14)|(s15|s16|s17);       //  断面線を作成
23:     Surface p1=revolve_z(s1,24);                        //  ポリゴンモデル
24:
25:     //  とってのの作成
26:     #define     R3      25.                             //  とっての半径
27:     #define     R4      15.                             //  とっての太さ
28:
29:     Line    s21=R4*(Point(-.2,0,1)|Point(.4,0,0)|       //  とっての断面
30:                     0.3*circle_xz(2,  0, 90)|Point(0,0,-1.4)|
31:                     0.3*circle_xz(2, 90,180)|Point(-.4,0,0)|
32:                     0.3*circle_xz(2,180,270)|Point(0,0,1.4)|
33:                     0.3*circle_xz(2,270,360));
34:     Line    s22=R3*(Point(1.5,0,0)|Point(0,1,0)|        //  とっての曲がり
35:                     circle_xy(4, 0, 90)|Point(-1,0,0)|
36:                     circle_xy(4,90,180)|Point(0,-1,0));
37:     Surface p2=move(0,R0+R2,L1/2+R1+T)*rot_y(90)*sweep_xz(s21,s22);
38:
39:     Line    s3=Line(Point(0,0,0),Point(180,0,0))|       //  テーブル
40:                     10*circle_xz(8, 0,180);
41:     Surface p3=move(0,0,-50)*revolve_z(s3,8);
42:     Attri   At3=Attri(0.5*WHITE+0.4*RED, 0.7,0.7,0.3);
43:     TMatrix     m=move(50,50,-50+1);
44:
45:     //  レンダリング
46:     Attri       At=Attri(WHITE*0.8+YELLOW*0.2,0.7,0.7,0.3);
47:     Scene       sc;
48:     sc.regist( m*p1, At );
49:     sc.regist( m*p2, At );
50:     for(int i=0; i<8; i++)
51:         sc.regist ( m*move(-50,-50,-1)*revolve_z(s3,1,360*i/8.,360*(i+1)/8.), At3);
52:
53:     Color   bc=WHITE*0.3+GREEN*0.1+BLUE*0.2;
54:     Win3D   w1("AP2_2w1",bc),w2("AP2_2w2",bc),w3("AP2_2w3",bc),w4("AP2_2w4",bc);
55:     w1.render( sc, POLYGON, WIRE|NOS);
56:     w2.render( sc, POLYGON, FLAT);
57:     w3.render( sc, POLYGON, GURO);
58:     w4.render( sc, POLYGON, PHONG);
59:     pause("AP2_2");
60: }
```

リスト　AP2_3.cpp

```
 1: //  多面球体を描画する（アンチエイリアシング）
 2: #include        "graph4.h"
```

```
 3:
 4: Surface div_polygon(Polygon& a, float r){
 5:         Point   t[6];
 6:         t[0]=a.p[0];              t[2]=a.p[1];            t[4]=a.p[2];
 7:         t[1]=r*unit(t[0]+t[2]); t[3]=r*unit(t[2]+t[4]); t[5]=r*unit(t[4]+t[0]);
 8:         return Polygon(t[0],t[1],t[5])||Polygon(t[1],t[2],t[3])
 9:             || Polygon(t[3],t[4],t[5])||Polygon(t[1],t[3],t[5]);
10: }
11:
12: main()
13: {
14:     #define     N       5               //  N=3:K=4 N=4:K=8 N=5:K=20
15:     #define     M       1               //  K*4^M 個の三角形
16:     #define     R       100             //  半径
17:
18:     int     n=N;
19:     float   a=sqrt(4*sin(PI/n)*sin(PI/n)-1);
20:     float   r=(1+a*a)/(2*a);
21:     Vector  v=Vector(1,0,r-a)/r;
22:
23:     Surface g[10];
24:     g[0]=revolve_z(Point(0,0,1)||v, 1, 0, 360./n );
25:     for(int i=0; i<M; i++)
26:         for(int j=0; j<g[i].n; j++)
27:                 g[i+1]=g[i+1]||div_polygon(g[i].p[j],1);
28:
29:     TMatrix e=rotate(unit(v),360.*(n-1)/n);
30:     Surface gg;
31:     for( i=0; i<n; i++)
32:         if( N<4 && i!=0 )gg=gg||(rot_z(360.*i/n)*g[M]);
33:         else              gg=gg||(rot_z(360.*i/n)*(g[M]||e*g[M]));
34:     if(N>4) gg=gg||rot_y(180)*gg;
35:      gg= scale( R, R, R )*gg;
36:
37:     Scene   sc1,    sc2;
38:     Attri   At=Attri(WHITE,0.7,0.5,0.3);
39:     sc1.regist( gg, At, FLAT );
40:     Win3D   w1("AP2_3w1",280,245,WHITE*0.2+BLUE*0.3);
41:     w1.setview( 60, 10, 1000);
42:     w1.render ( sc1, POLYGON, FLAT );
43:
44:     sc2.regist( scale(2,2,2)*gg, At, FLAT );
45:     Win3D   w20("AP2_3",560,490,WHITE*0.2+BLUE*0.3);
46:     w20.setview( 60, 10, 2000);
47:     w20.render ( sc2, POLYGON, FLAT );
48:
49:     //  アンチエイリアシング画像
50:     Win3D   w2("AP2_3w2",280,245,WHITE*0.2+BLUE*0.3);
51:     for(int x=0; x<300; x++)
52:         for(int y=0; y<300; y++){
53:             w2.w.color( (w20.w.pixel(2*x,2*y)+w20.w.pixel(2*x+1,2*y)
54:                         +w20.w.pixel(2*x,2*y+1)+w20.w.pixel(2*x+1,2*y+1))/4 );
55:             w2.w.point(x,y);
56:         }
57:     pause("AP2_3");
58: }
```

リスト　AP3_1.cpp

```
 1: //  レイトレーシング法
 2: //  ２次曲面を表示する
 3:
 4: #include        "graph4.h"
 5: #include        "ray.h"
 6:
 7: //  ２次曲面のクラス
 8: class   Hyper{
```

```
 9:         public:
10:         Vector  S2,S1;                                  // 係数
11:         float   s0;
12:         Hyper(Vector s2i,Vector s1i,float s0i){ S2=s2i; S1=s1i; s0=s0i;    }
13:         float   hit(Ray& VR, Point& P, Vector& N);
14: };
15:
16: // 視線が２次曲面と交わるかのチェック（交点座標Ｐを返す）
17: float   Hyper::hit(Ray& VR, Point& P, Vector& N){
18:         Point   V0=VR.o;
19:         Vector  V= VR.d;
20:         float   d=((scale(S2)*V)*V0+.5*S1*V)*((scale(S2)*V)*V0+.5*S1*V) // 判別式
21:                 -((scale(S2)*V)*V)*((scale(S2)*V0)*V0+S1*V0+s0);
22:         if( d < 0 ) return INFINITY;
23:         float   ad=sqrt(d)/fabs((scale(S2)*V)*V);
24:         float   t=-((scale(S2)*V)*V0+.5*S1*V)/((scale(S2)*V)*V)-ad;     // 距離
25:         P=VR.o+t*VR.d;
26:         if(t<0||P.x<-120||120<P.x||P.y<-120||120<P.y||P.z<-120||120<P.z)t=t+2*ad;
27:         P=VR.o+t*VR.d;
28:         if(t<0||P.x<-120||120<P.x||P.y<-120||120<P.y||P.z<-120||120<P.z)return INFINITY;
29:         N=unit(2*scale(S2)*P+S1);
30:         return t;
31: }
32:
33: // 直線を描画します
34: void    line(Win3D& w, Hyper& Hy, Point& p0, Point& p1, Color& c)   {
35:         Point   a0=w.screen(p0),a1=w.screen(p1),P;
36:         int     xs=(a1.x-a0.x>0)?1:-1;
37:         int     ys=(a1.y-a0.y>0)?1:-1;
38:         if(abs(a0.x-a1.x)>abs(a0.y-a1.y))
39:             for(int x=a0.x+.5; x!=(int)(a1.x+.5); x+=xs){
40:                 Point   B=(a0+(a1-a0)*(x-a0.x)/(a1.x-a0.x));
41:                 Point   A=w.world(B);
42:                 Vector  V=unit(A-w.v0),N;
43:                 Ray     VR=Ray(w.v0,V);                 // 視線の記述
44:                 if(Hy.hit(VR,P,N)>len(A-w.v0)){ w.color(c); w.point(x,B.y); }
45:                 else{  w.color(.5*c+w.w.pixel(x,B.y));  w.point(x,B.y); }
46:             }
47:         else for(int y=a0.y-.5; y!=(int)(a1.y-.5); y+=ys){
48:                 Point   B=(a0+(a1-a0)*(y-a0.y)/(a1.y-a0.y));
49:                 Point   A=w.world(B);
50:                 Vector  V=unit(A-w.v0),N;
51:                 Ray     VR=Ray(w.v0,V);                 // 視線の記述
52:                 if(Hy.hit(VR,P,N)>len(A-w.v0)){ w.color(c); w.point(B.x,y); }
53:                 else{  w.color(.5*c+w.w.pixel(B.x,y));  w.point(B.x,y); }
54:             }
55: }
56:
57: main()
58: {
59:     float   a=40,   b=40,   c=50,   s0=0.0;
60:
61:     Vector  S2( 1/(a*a),  1/(b*b),  1/(c*c)), S1(0,0,0);  s0=-1;  // 楕円面
62: //  Vector  S2( 1/(a*a),  1/(b*b), -1/(c*c)), S1(0,0,0);  s0=-1;  // 一葉双曲面
63: //  Vector  S2(-1/(a*a), -1/(b*b),  1/(c*c)), S1(0,0,0);  s0=-1;  // 二葉双曲面
64: //  Vector  S2( 1/(a*a),  1/(b*b), -1/(c*c)), S1(0,0,0);  s0=0;   // 楕円錐面
65: //  Vector  S2( 1/(a*a), -1/(b*b),  0),    S1(0,0,-1/c);  s0=0;   // 楕円放物面
66: //  Vector  S2( 1/(a*a),  1/(b*b),  0),    S1(0,0,-1/c);  s0=0;   // 双曲放物面
67: //  Vector  S2( 1/(a*a),  0,        0),    S1(0,1/b,0);   s0=0;   // 放物柱面
68: //  Vector  S2( 1/(a*a),  1/(b*b),  0),       S1(0,0,0);  s0=-1;  // 楕円柱面
69: //  Vector  S2( 1/(a*a), -1/(b*b),  0),       S1(0,0,0);  s0=-1;  // 双曲柱面
70:
71:     Hyper   Hy=Hyper( S2, S1, s0 );                     // ２次曲面の記述
72:     Color   C=0.5*RED, bc=0.2*WHITE;
73:
74:
75:     Win3D   w("AP3_1",bc);
76:     w.setview(60,30,3000);
```

```
 77:     Point   V0=w.v0;                                    //  視点の記述
 78:     Point   L0=Point(100,200,000);                      //  照明光の記述
 79:
 80:     Vector  V=unit(Point(0,0,1)-V0),N;
 81:     Ray     VR=Ray(V0,V);                               //  視線の記述
 82:     Point   P;                                          //  交点座標
 83:     if(Hy.hit(VR,P,N)>=INFINITY){
 84:         if(2*Hy.S2.z+Hy.S1.z>0)
 85:         {  Hy.S2=-Hy.S2;   Hy.S1=-Hy.S1;   Hy.s0=-Hy.s0;   }
 86:         else if( (2*Hy.S2.z+Hy.S1.z==0)&&(2*Hy.S2.y+Hy.S1.y>0))
 87:         {  Hy.S2=-Hy.S2;   Hy.S1=-Hy.S1;   Hy.s0=-Hy.s0;   }
 88:         else if( ((2*Hy.S2.z+Hy.S1.z)*(2*Hy.S2.y+Hy.S1.y)==0)&&(2*Hy.S2.x+Hy.S1.x>0))
 89:         {  Hy.S2=-Hy.S2;   Hy.S1=-Hy.S1;   Hy.s0=-Hy.s0;   }
 90:     }
 91:     if( Hy.hit(VR,N,P)<INFINITY && N*V>0)               //  視線が２次曲面と交わるかの判定
 92:     {  Hy.S2=-Hy.S2;   Hy.S1=-Hy.S1;   Hy.s0=-Hy.s0;   }
 93:
 94:     for(int x=0; x<w.size_x; x++)
 95:         for(int y=0; y<w.size_y; y++){
 96:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);// 視線ベクトルの計算
 97:             Ray     VR=Ray(V0,V);                       //  視線の記述
 98:
 99:             if(Hy.hit(VR,P,N)<INFINITY){                //  視線が２次曲面と交わるかの判定
100:                 Vector  L=unit(P-L0);                   //  照明光の方向ベクトル
101:                 w.color( shading(V,L,N,C)+w.w.pixel(x,y) );
102:                 w.point(x,y);
103:                 if(Hy.hit(Ray(P+V,V),P,N)<INFINITY){
104:                     Vector  L=unit(P-L0);               //  照明光の方向ベクトル
105:                     w.color( shading(V,L,N,C,0)+w.w.pixel(x,y) );
106:                     w.point(x,y);
107:                 }
108:             }
109:     }
110:
111:     //  座標軸
112:     line( w, Hy, Point(-180,0,0), Point(180,0,0), CYAN);    w.w.gprintf("X");
113:     line( w, Hy, Point(0,-180,0), Point(0,180,0), CYAN);    w.w.gprintf("Y");
114:     line( w, Hy, Point(0,0,-180), Point(0,0,190), CYAN);    w.w.gprintf("Z");
115:     pause("AP3_1");
116: }
```

リスト AP3_2.cpp

```
  1: //  物体（球体、床）を複数描画する（レイトレーシング法）
  2:
  3: #include        "graph4.h"
  4: #define         NUM_REF 8                               //  多重反射の回数
  5:
  6: main()
  7: {
  8:     Scene   sc(0.25*CYAN);                              //  複数物体（球体、壁）の記述
  9:     Attri   Au=Attri( RED,     .7,.7,.3, .2);
 10:     Attri   Ad=Attri( YELLOW,  .7,.7,.3, .1);
 11:     Attri   Aw=Attri( .2*WHITE+.1*BLUE, .7,.7,.3, .7);
 12:
 13:     //  球体の記述（2段重ね）
 14:     #define NUM_LB  5                                   //  下の球体の個数
 15:     float   r2=70,r1=40;                                //  上の球体,下の球体の半径
 16:     float   r0=r1/sin(PI/NUM_LB);                       //  下の球体を配置する半径
 17:     Point   A=  Point(0,0,-50);                         //  下の球体配置の中心
 18:     float   h2=sqrt((r2+r1)*(r2+r1)-r0*r0)+1;
 19:     sc.regist( Ball(A+h2*Point(0,0,1),r2), Au );        //  上の球体
 20:     for(int k=0; k<NUM_LB; k++)                         //  下の球体
 21:         sc.regist( Ball(A+r0*(rot_z(360.*k/NUM_LB)*Point(0,1,0)),r1), Ad );
 22:
 23:     //  床、壁の記述（Wa：ポリゴン）
```

```
24:     #define     LEN     120
25:     Polygon po[3];
26:     po[0]=Polygon(  Point( LEN, LEN,A.z-r1),Point(-LEN,LEN,A.z-r1),
27:                     Point(-LEN,-LEN,A.z-r1),Point(LEN,-LEN,A.z-r1));
28:     po[1]=Polygon(  Point( LEN,-LEN,100),   Point( LEN,-LEN,A.z-r1),
29:                     Point(-LEN,-LEN,A.z-r1),Point(-LEN,-LEN,100));
30:     po[2]=Polygon(  Point(-LEN, LEN,100),   Point(-LEN,-LEN,100),
31:                     Point(-LEN,-LEN,A.z-r1),Point(-LEN, LEN,A.z-r1));
32:     for( k=0; k<3; k++) sc.regist( Wall( po[k]), Aw );
33:
34:     Win3D  w("AP3_2");
35:     w.setview ( 60, 30 );                          //  照明光の記述
36:     w.setlight( 60, 45 );                          //  照明光の記述
37:     w.render( sc, RAY , NUM_REF);
38:     pause("AP3_2");
39: }
```

リスト AP3_3.cpp

```
 1: //  レイトレーシング法
 2: //  物体（球体、床）を複数にする（影,多重反射,屈折の処理も行う）
 3:
 4: #include       "graph4.h"
 5: #define        NUM_REF 8                          //  多重反射の回数
 6:
 7: main()
 8: {
 9:     Scene   sc(0.25*CYAN);                        //  複数物体（球体、壁）の記述
10:     Attri   Au=Attri( BLACK,   .7,.7,.3, .2,1.5);
11:     Attri   Ad=Attri( YELLOW,  .7,.7,.3, .1);
12:     Attri   Aw=Attri( .2*WHITE+.1*BLUE, .7,.7,.3, .7);
13:
14:     //  球体の記述（2段重ね）
15:     #define NUM_LB  5                             //  下の球体の個数
16:     float   r2=70,r1=40;                          //  上の球体,下の球体の半径
17:     float   r0=r1/sin(PI/NUM_LB);                 //  下の球体を配置する半径
18:     Point   A= Point(0,0,-50);                    //  下の球体配置の中心
19:     float   h2=sqrt((r2+r1)*(r2+r1)-r0*r0)+1;
20:     sc.regist( Ball(A+h2*Point(0,0,1),r2), Au );  //  上の球体
21:     for(int k=0; k<NUM_LB; k++)                   //  下の球体
22:         sc.regist( Ball(A+r0*(rot_z(360.*k/NUM_LB)*Point(0,1,0)),r1), Ad );
23:
24:     //  床、壁の記述（Wa：ポリゴン）
25:     #define     LEN     120
26:     Polygon po[3];
27:     po[0]=Polygon(  Point( LEN, LEN,A.z-r1),Point(-LEN,LEN,A.z-r1),
28:                     Point(-LEN,-LEN,A.z-r1),Point(LEN,-LEN,A.z-r1));
29:     po[1]=Polygon(  Point( LEN,-LEN,100),   Point( LEN,-LEN,A.z-r1),
30:                     Point(-LEN,-LEN,A.z-r1),Point(-LEN,-LEN,100));
31:     po[2]=Polygon(  Point(-LEN, LEN,100),   Point(-LEN,-LEN,100),
32:                     Point(-LEN,-LEN,A.z-r1),Point(-LEN, LEN,A.z-r1));
33:     for( k=0; k<3; k++) sc.regist( Wall( po[k]), Aw );
34:
35:     Win3D   w1("AP3_3w1"),w2("AP3_3w2");
36:     w1.setview ( 30, 45 );                        //  照明光の記述
37:     w1.setlight( 60, 45 );                        //  照明光の記述
38:     w1.render (sc, RAY, NUM_REF);
39:
40:     w2.setview ( 60, 30 );                        //  照明光の記述
41:     w2.setlight( 60, 45 );                        //  照明光の記述
42:     w2.render (sc, RAY, NUM_REF);
43:
44:     pause("AP3_3");
45: }
```

リスト AP3_4.cpp

```
 1: //  レンズを描画にする
 2:
 3: #include        "graph4.h"
 4: #define         NUM_REF 8                          //  視線衝突・通過制限(>0)
 5:
 6: main()
 7: {
 8:     Scene   s1("..¥¥pic¥¥snow.bmp");               //  複数物体（球体、壁）の記述
 9:     Scene   s2("..¥¥pic¥¥snow.bmp");               //  複数物体（球体、壁）の記述
10:     float   da=60,  de=0;                          //  レンズの方向
11:
12:     //  レンズの記述
13:     Attri   Al=Attri(BLACK,0.7,0.7,0.3, 0.1,1.7);
14:     TMatrix dir=rot_z(da)*rot_y(90-de);            //  レンズの回転行列
15:
16:     Csgm    H1=Csgm(Point(1,1,1),Point(0,0,0),-150*150);//  楕円
17:     Csgm    H2=Csgm(Point(1,1,0),Point(0,0,0),-100*100);//  円柱
18:     Csgm    C0=(move(0,0,100)*H1)*(move(0,0,-100)*H1); //  凸レンズ
19:     Csgm    C1=dir*( plane_xz()*H2*C0 );
20:     s1.regist( C1 , Al, Ball(Point(0,0,0),120) );
21:
22:     Csgm    H3=Csgm(Point(-1,-1,20),Point(0,0,0),-40*40);// 二葉
23:     Csgm    C2=dir*( plane_xz()*H2*H3 );               //  凹レンズ
24:     s2.regist( C2 , Al, Ball(Point(0,0,0),120) );
25:
26:     Win3D   w1("AP3_4w1"),w2("AP3_4w2");
27:     w1.setview  (200*unit(da+12,de));              //  視点０の記述
28:     w1.setlight(Point(200,200,500));               //  照明光の記述
29:     w1.render   (s1, RAY, NUM_REF);
30:
31:     w2.setview  (200*unit(da+12,de));              //  視点１の記述
32:     w2.setlight(Point(200,200,500));               //  照明光の記述
33:     w2.render   (s2, RAY, NUM_REF);
34:
35:     pause("AP3_4");
36: }
```

リスト AP3_5.cpp

```
 1: //  テーブルと灰皿
 2:
 3: #include        "graph4.h"
 4: #define         NUM_REF 10                         //  視線衝突・通過制限(>0)
 5: #define         L       -70                        //  床の高さ（位置）
 6:
 7: main()
 8: {
 9:     Scene   sc(WHITE*0.3);
10:
11:     //  テーブルの記述
12:     #define     H       100                        //  テーブルの高さ
13:     Point   A=Point(50,50,0),    B=A+Point(0,0,L+H/2);
14:     Attri   Al=Attri(BLACK, 0.7,0.7,0.3, 0.1,1.5);
15:     Csgm    T0=ball(Point(70,70,15));              //  上板
16:     Csgm    T1=  move(rot_z(  0)*Point(20,0,0))*T0
17:                 +move(rot_z(120)*Point(20,0,0))*T0
18:                 +move(rot_z(240)*Point(20,0,0))*T0;
19:     sc.regist( move(A)*move(0,0,L+H+5)*(T1*board_xy(10)) , Al );
20:
21:     Attri   At=Attri(0.4*YELLOW, .7,.9,.3, .5);
22:     Csgm    T2=ball(Point(20,20,H/2))*ball(Point(8,8,0));// 足
23:     sc.regist( move(B)*move(rot_z(  0)*Point(60,0,0))*T2 , At );
24:     sc.regist( move(B)*move(rot_z(120)*Point(60,0,0))*T2 , At );
```

```
25:     sc.regist( move(B)*move(rot_z(240)*Point(60,0,0))*T2 , At );
26:
27:     //  灰皿の記述
28:     Attri   Ah=Attri(BLACK, 0.7,0.7,0.3, 0.3,2.5,0.90);
29:     Csgm    H1=ball(Point(30,30,0))*board_xy(20);
30:     Csgm    H2=ball(Point(20,20,0))*ball(Point(40,40,15));
31:     Csgm    H3=H1-move(0,0,10)*H2-move(0,0,10)*ball(Point(7,0,5));
32:     sc.regist( move(A)*move(30,30,L+H+10+11)*H3, Ah,
33:                         Ball(A+Point(30,30,L+H+10+20),35) );
34:
35:     //  床、壁の記述 (Wa : ポリゴン)
36:     #define     LEN     130
37:     Polygon po[3];
38:     po[0]=Polygon(  Point( LEN, LEN,L  ),Point(-LEN,LEN,L  ),
39:                     Point(-LEN,-LEN,L  ),Point(LEN,-LEN,L  ));
40:     po[1]=Polygon(  Point(-LEN,-LEN,100),Point( LEN,-LEN,100),
41:                     Point( LEN,-LEN,L  ),Point(-LEN,-LEN,L  ));
42:     po[2]=Polygon(  Point(-LEN, LEN,100),Point(-LEN,-LEN,100),
43:                     Point(-LEN,-LEN,L  ),Point(-LEN, LEN,L  ));
44:
45:     Attri   Aw("..¥¥pic¥¥wall.bmp"  ,0.7,0.7,0.3,0);
46:     Attri   Af("..¥¥pic¥¥floor.bmp" ,0.5,0.7,0.3,0.3);
47:     sc.regist( Wall( po[0]), Af);
48:     sc.regist( Wall( po[1]), Aw);
49:     sc.regist( Wall( po[2]), Aw);
50:
51:     //  描画
52:     Win3D   w1("AP3_5w1"),w2("AP3_5w2");
53:     w1.setview ( 60, 40 );                          //  視点０の記述
54:     w1.render  ( sc, RAY, NUM_REF);
55:     w2.setview ( 30, 30 );                          //  視点１の記述
56:     w2.render  ( sc, RAY, NUM_REF);
57:     pause("AP3_5");
58: }
```

リスト　AP4_1.cpp

```
 1: //  レイトレーシング法
 2: //  コイル・スプリングを描画する
 3:
 4: #include        "graph4.h"
 5:
 6: main()
 7: {
 8:     //  モデリング
 9:     //  コイルの作成
10:     #define     R0      50.                         //  コイルの巻き半径
11:     #define     R1      15.                         //  コイルの半径
12:     #define     RN      4                           //  巻き数
13:     #define     L1      60.                         //  一巻きの長さ
14:     #define     DEV0    16                          //  巻きの分割数
15:     #define     DEV1    8                           //  断面の分割数
16:
17:     Line    s1=R1*circle_xz( DEV1);                 //  コイルの断面
18:     Line    s2=Line(DEV0*RN+1);                     //  コイルを巻く軌跡
19:     for(int t=0; t<DEV0*RN+1; t++)
20:         s2.p[t]=rot_z(t*360./DEV0)*Point(R0,0,L1*t/DEV0);
21:     Surface p1=sweep_xz( s1, s2 );                  //  表面ポリゴン
22:
23:     //  端部品の作成
24:     Line    s3=circle_xz( DEV1/2, 0, 90);           //  円弧の生成
25:     Surface a1=rot_x(-90)*rot_z(-90)*revolve_z(R1*s3,DEV1 );//  表面ポリゴン
26:     Surface p2=move(startpoint(s2))*rot_xz(startvector(s2))*a1;
27:     Surface p3=move(endpoint(s2))*rot_xz(endvector(s2))*a1;
28:     TMatrix m=move(-100*unit(30,30))*move(0,0,-130);
29:
```

```
30:     // レンダリング
31:     Attri   A1=Attri(RED,   .0,.7,.0, .8);
32:     Scene   sc1("..¥¥pic¥¥tree.bmp",1, 0.7);
33:     sc1.regist( m*p1, A1 );
34:     sc1.regist( m*p2, A1 );
35:     sc1.regist( m*p3, A1 );
36:     Win3D   w1("AP4_1w1");
37:     w1.setview( 30, 30, 700 );
38:     w1.render ( sc1, RAY, 10);
39:
40:     Attri   A2=Attri(RED,   .0,.7,.0, .1,1.5,.9);       // 屈折率1.5
41:     Scene   sc2("..¥¥pic¥¥tree.bmp",1, 0.7);
42:     sc2.regist( m*p1, A2 );
43:     sc2.regist( m*p2, A2 );
44:     sc2.regist( m*p3, A2 );
45:     Win3D   w2("AP4_1w2");
46:     w2.setview( 30, 30, 700 );
47:     w2.render ( sc2, RAY, 4);
48:
49:     pause("AP4_1");
50: }
51:
52:
```

リスト AP4_2.cpp

```
 1: // レイトレーシング法
 2: #include        "graph4.h"
 3:
 4: main()
 5: {
 6:     // モデリング
 7:     // 胴体の作成
 8:     #define     R0      60.                             // カップの内径
 9:     #define     R1      25.                             // カップ底の曲率
10:     #define     T       10.                             // カップの厚さ
11:     #define     H       150.                            // カップの高さ
12:     #define     R2      (T/2.)
13:     #define     L1      (H-T-R1-R2)
14:
15:     Line    s11=Point(0,0,T)|Point(R0-R1,0,0);          // 底       (内側)
16:     Line    s12=R1*circle_xz(2,180,90);                 // 底の丸み (内側)
17:     Line    s13=Point(0,0,0)|Point(0,0,L1);             // 側面     (内側)
18:     Line    s14=R2*circle_xz(5,-90,90);                 // 上部
19:     Line    s15=Point(0,0,0)|Point(0,0,-L1);            // 側面     (外側)
20:     Line    s16=(R1+2*R2)*circle_xz(3,90,180);          // 底の丸み (外側)
21:     Line    s17=Point(0,0,0)|Point(-R0+R1,0,0);         // 底       (外側)
22:     Line    s1=(s11|s12)|(s13|s14)|(s15|s16|s17);       // 断面線を作成
23:     Surface p1=revolve_z(s1,13);                        // ポリゴンモデル
24:
25:     // とっての作成
26:     #define     R3      25.                             // とっての半径
27:     #define     R4      15.                             // とっての太さ
28:     Line    s21=R4*(Point(-.2,0,1)|Point(.4,0,0)|       // とっての断面
29:                     0.3*circle_xz(2,  0, 90)|Point(0,0,-1.4)|
30:                     0.3*circle_xz(2, 90,180)|Point(-.4,0,0)|
31:                     0.3*circle_xz(2,180,270)|Point(0,0,1.4)|
32:                     0.3*circle_xz(2,270,360));
33:     Line    s22=R3*(Point(1.5,0,0)|Point(0,1,0)|        // とっての曲がり
34:                     circle_xy(4,  0, 90)|Point(-1,0,0)|
35:                     circle_xy(4,90,180)|Point(0,-1,0));
36:     Surface p2=move(0,R0+R2,L1/2+R1+T)*rot_y(90)*sweep_xz(s21,s22);
37:
38:     Line    s3=Line(Point(0,0,0),Point(180,0,0))|       // テーブル
39:                             10*circle_xz(4,0,180);
40:     Attri   At3=Attri(0.5*WHITE+0.3*RED, 0.6,0.7,0.3);
```

```
41:     TMatrix m=move(-200*unit(30,30))*move(50,50,-120)
42:                     *rotate(unit(30,0),unit(30,10));
43:
44:     //  レンダリング
45:     Attri   A1=Attri(WHITE,0.0,0.7,0.0, .75);
46:     Scene   sc1("..¥¥pic¥¥tree.bmp",1, 0.80);
47:     sc1.regist ( m*p1, A1 );
48:     sc1.regist ( m*p2, A1 );
49:     for(int i=0; i<8; i++)
50:         sc1.regist ( m*move(-50,-50,-1)*revolve_z(s3,1,360*i/8.,360*(i+1)/8.), At3);
51:     Win3D   w1 ("AP4_2w1");
52:     w1.setview ( 30, 30, 700 );
53:     w1.render  ( sc1, RAY, 20);
54:
55:     Attri   A2=Attri(WHITE, 0.0,0.7,0.0, 0.1,1.5);        //  屈折率1.5
56:     Scene   sc2("..¥¥pic¥¥tree.bmp",1, 0.80);
57:     sc2.regist ( m*p1, A2 );
58:     sc2.regist ( m*p2, A2 );
59:     for( i=0; i<8; i++)
60:         sc2.regist ( m*move(-50,-50,-1)*revolve_z(s3,1,360*i/8.,360*(i+1)/8.), At3);
61:     Win3D   w2 ("AP4_2w2");
62:     w2.setview ( 30, 30, 700 );
63:     w2.render  ( sc2, RAY, 4);
64:     pause("AP4_2");
65: }
```

リスト AP4_3.cpp

```
 1: //  ミラーボールを描画する
 2:
 3: #include        "graph4.h"
 4:
 5: Surface div_polygon(Polygon& a, float r){
 6:         Point   t[6];
 7:         t[0]=a.p[0];    t[2]=a.p[1];    t[4]=a.p[2];
 8:         t[1]=r*unit(t[0]+t[2]);
 9:         t[3]=r*unit(t[2]+t[4]);
10:         t[5]=r*unit(t[4]+t[0]);
11:         return Polygon(t[0],t[1],t[5])||Polygon(t[1],t[2],t[3])
12:             || Polygon(t[3],t[4],t[5])||Polygon(t[1],t[3],t[5]);
13: }
14:
15: main()
16: {
17:     //  形状の作成（モデリング）
18:     #define     N       5               //  N=3:K=4 N=4:K=8 N=5:K=20
19:     #define     M       2               //  K*4^M 個の三角形(ミラー数)
20:     #define     R       100             //  半径
21:
22:     int     n=N;
23:     float   a=sqrt(4*sin(PI/n)*sin(PI/n)-1);
24:     float   r=(1+a*a)/(2*a);
25:     Vector  v=Vector(1,0,r-a)/r;
26:
27:     Surface g[10];
28:     g[0]=revolve_z(Point(0,0,1)||v, 1, 0, 360./n );
29:     for(int i=0; i<M; i++)
30:         for(int j=0; j<g[i].n; j++)
31:                 g[i+1]=g[i+1]||div_polygon(g[i].p[j],1);
32:
33:     TMatrix e=rotate(unit(v),360.*(n-1)/n);
34:     Surface gg;
35:     for( i=0; i<n; i++)
36:         if( N<4 && i!=0 )gg=gg||(rot_z(360.*i/n)*g[M]);
37:         else            gg=gg||(rot_z(360.*i/n)*(g[M]||e*g[M]));
38:     if(N>4) gg=gg||rot_y(180)*gg;
```

```
39:      gg= scale( R, R, R )*gg;
40:
41:     // 描画（レンダリング）
42:     // ミラーボール
43:     Scene   s1("..\\pic\\snow.bmp");
44:     Attri   At=Attri(WHITE*0,0.7,0.3,0.3, .9);
45:     s1.regist( gg, At, FLAT );
46:     Win3D   w1("AP4_3w1",500,500);
47:     w1.setview( 60, 10, 200);
48:     w1.render ( s1, RAY, 6 );
49:
50:     // ガラスボール
51:     Scene   s2("..\\pic\\snow.bmp");
52:     Attri   Ac=Attri(WHITE*0 ,0.7,0.3,0.3, .2,1.5);
53:     s2.regist( gg, Ac, FLAT );
54:     Win3D   w2("AP4_3w2",500,500);
55:     w2.setview( 60, 10, 200);
56:     w2.render ( s2, RAY, 20 );
57:
58:     pause("AP4_3");
59: }
```

# 演習問題の解答例

演習の解答例は，プロジェクト"PR"に格納されています．

3-1　任意点における回転移動（PR3_1）

Y字型図形を，点（0，－100，0）を中心に$X$軸回転（30，60，90度）せよ．

・説明

中心位置を指定しての回転移動は，以下のような手順で変換を行うことができます．

① 平行移動変換を行い，回転位置（0，－100，0）を原点に移動する．

② 回転変換を行う（$X$軸回転）．

③ 平行移動変換を行い，原点をもとの位置に戻す．

すなわち，任意の位置での変換は，①回転位置を原点に平行移動して，②原点で目的の変換を行い，③平行移動でもとの位置に戻すことにより行うことができます．

プログラム例および実行結果を示します．

3-2　らせん状の回転移動

Y字型図形を，$Y$軸方向に50移動する．その図形を$Z$軸回転（$360t/N$度）し，$Z$軸方向に$100t/N$移動せよ（$N$=36，$t$=$-N$～$N$，複数の図形を描画する）．

・説明

プログラム例および実行結果を示します．

3-3　らせん状の回転移動

$YZ$平面上の長方形（60×40）を，以下のように変換せよ．

① $X$軸回転する（$360t/N$度）．

② $Y$軸方向に130移動する．

③ $Z$軸回転する（$360t/N$度）．

$t$=0～$N-1$，$N$=150とする．

・説明

プログラム例および実行結果を示します．

5-1　プログラム例EX5_3.cppを変更して，半分の球体を描画せよ（不自然な点を観察せよ）．

（main内を，以下のように変更する）

```
Surface  f=revolve_z( 100*a, 18, 90, 270 );
```

・説明

実行結果を示します．球体の内側のポリゴンが表示されません．

（実行例では，球体の内側のポリゴンをワイヤーフレーム表示しています）

5-2　プログラム例EX5_4.cppを変更して，プログラム例EX5_5.cppのようなトーラスと球体が重なるようすを，$Z$ソートで描画せよ．$Z$ソートでは，うまく描画できないことを確認せよ．

・説明

トーラスと球体を構成するポリゴンの透視変換後の$Z$座標でソートし，視点から遠い順にポリゴンを描画します．その際，描画するポリゴンが，トーラスに属するか球体に属するか区別して，物体の色を指定する必要があります．そのため，球体のポリゴンIDには1000加えて，トーラスのポリゴンと区別しています．

プログラム例および実行結果を示します．

6-1　プログラム例EX6_1.cpp，EX6_2.cpp，EX6_3.cppを変更して，ポリゴン数を少なくした場合について違いを観察せよ(ポリゴン数は，main内のcircle_xz，rezolve_z関数の引数で変わります)．

・説明

実行結果を口絵に示します．

6-2　プログラム例EX6_3.cppを変更して，式(6-5)中のスペキュラー（鏡面反射光）成分の項$k_s \cos^n \beta \cdot C_W$の$n$を変更（5，10，20，40）した場合の違いを観察せよ．また，スペキュラー成分のないランバーモデルについても観察せよ．

（$n$=5とする場合には，shading関数内を以下のように変更する）

```
return  kd*max(-N*L,0)*c+ks*pow(max(-R*V,0),5)*WHITE+ke*c;
```

・説明

実行結果を口絵に示します．

7-1　プログラム例EX7_3.cppを変更して，球体（映り込みのない）と映り込みのある床を描画せよ．

・説明

床の描画で反射視線を考え，それが球体と交わる場合に球体の反射を考慮します．プログラム例および実行結果を示します．

7-2　プログラム例EX7_3.cppおよびEX7_4.cppの反射率及屈折率を変更して，画像の変化を観察せよ．

・説明

EX7_3.cppを変更して，反射率を0.6にした場合の実行結果を示します．

7-3　プログラム例EX7_6.cppを下記のように変更して，バウンディングボリュームを使用しないで描画し，描画時間を比較せよ．

```
sc.regist( A1, Al, Ball(Point(0,0,25),110) );  ➔  sc.regist( A1, Al );
sc.regist( A3, Al, Ball(Point(0,0,25),40) );   ➔  sc.regist( A3, Al );
```

・説明

実行時間が表示されるので，比較して下さい．遅くなります．

8-1　プログラム例EX8_3.cppを変更して，球体にテクスチャーマッピングとバンプマッピングを行って描画せよ．マッピング画像は，"..¥¥pic¥¥texture2.bmp"，"..¥¥pic¥¥bump2.bmp"を使用せよ．

マッピング画像は，"..¥¥pic¥¥texture2.bmp"，"..¥¥pic¥¥bump2.bmp"を使用せよ．

・説明

描画色を計算するシェーディング関数の引数$C$（Color），$N$（Vector）を，それぞれプログラム例EX8_1.cpp，EX8_3.cppのようにします．プログラム例を示します．また，実行結果を口絵に示します．

リスト　PR3_1.cpp

```
 1: //       任意の位置を中心とする回転
 2:
 3: #include        "graph2.h"
 4:
 5: main()
 6: {
 7:     TMatrix t;                                              //  変換行列
 8:     Line    l=Line( Point(0,50,0),      Point(23,27,0),     //  Y字形の頂点座標の設定
 9:                     Point(23,0,0),      Point(33,0,0),
10:                     Point(33,27,0),     Point(50,50,0),
11:                     Point(40,50,0),     Point(30,36,0),
12:                     Point(16,50,0),     Point(0,50,0) );
13:
14:     Win3D   w1("PR3_1");                                    //  ウィンドウ生成
15:     w1.axis ();                                             //  座標軸の表示
16:     w1.color( CYAN );                                       //  描画色の設定
17:     w1.line ( l );                                          //  Y字形の描画
18:
19:     Point   a=Point(0,-100,0);                              //  回転中心
20:     t=move(a)*rot_x(30)*move(-a);                           //  変換行列の設定
21:     w1.color( RED );                                        //  描画色の設定
22:     w1.line ( t*l );                                        //  変換して描画
23:
24:     t=move(a)*rot_x(60)*move(-a);                           //  変換行列の設定
25:     w1.line ( t*l );                                        //  変換して描画
26:
27:     t=move(a)*rot_x(90)*move(-a);                           //  変換行列の設定
28:     w1.line ( t*l );                                        //  変換して描画
29:
30:     w1.color( BLUE );                                       //  描画色の設定
31:     w1.line ( Line( a+Point(200,0,0),
32:                     a+Point(-200,0,0) ) );                  //  回転軸の描画
33:
34:     pause("PR3_1");
35: }
```

リスト　PR3_2.cpp

```
 1: //       らせん状の回転移動
 2:
 3: #include        "graph2.h"
 4:
 5: #define         N   36
 6:
 7: main()
 8: {
 9:     TMatrix t;                                              //  変換行列
10:     Line    l=Line( Point(0,50,0),      Point(23,27,0),     //  Y字形の頂点座標の設定
11:                     Point(23,0,0),      Point(33,0,0),
12:                     Point(33,27,0),     Point(50,50,0),
13:                     Point(40,50,0),     Point(30,36,0),
14:                     Point(16,50,0),     Point(0,50,0) );
15:
16:     Win3D   w1("PR3_2");                                    //  ウィンドウ生成
17:     w1.axis ();                                             //  座標軸の表示
18:     l=move(Point(0,50,0))*l;
19:
20:     for(int i=-N; i<=N; i++){
21:         TMatrix t=move(Point(0,0,100.*i/N))*rot_z(360*i/N);
22:         w1.color( RED );                                    //  描画色の設定
23:         w1.line ( t*l );                                    //  変換して描画
24:     }
25:     w1.color( BLUE );                                       //  描画色の設定
26:     w1.line ( l );                                          //  Y字形の描画
```

```
27:
28:     pause("PR3_2");
29: }
```

リスト PR3_3.cpp

```
 1: //       らせん状の回転移動
 2:
 3: #include        "graph2.h"
 4:
 5: #define         N   150
 6:
 7: main()
 8: {
 9:     TMatrix t;                                          //  変換行列
10:     Line    l=Line( Point(0,20,40),Point(0,-20,40),     //  四角形
11:                     Point(0,-20,-40),Point(0,20,-40),Point(0,20,40));
12:
13:     Win3D   w1("PR3_3");                                //  ウィンドウ生成
14:     w1.axis ();                                         //  座標軸の表示
15:
16:     for(int i=0; i<N; i++){
17:         TMatrix t=rot_z(360.*i/N)*move(Point(0,130,0))*rot_x(360.*i/N);
18:         w1.color( RED );                                //  描画色の設定
19:         w1.line ( t*l );                                //  変換して描画
20:     }
21:     pause("PR3_3");
22: }
```

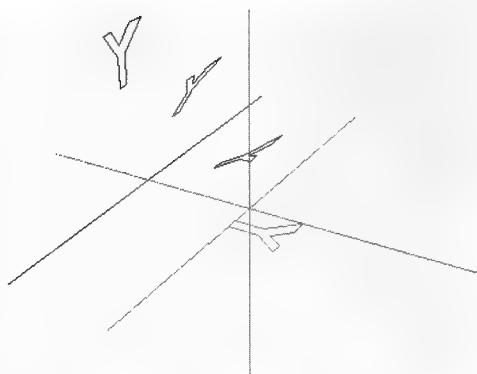
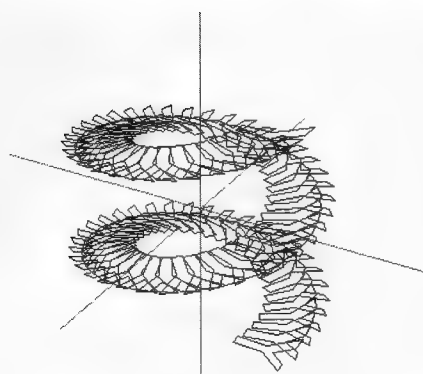
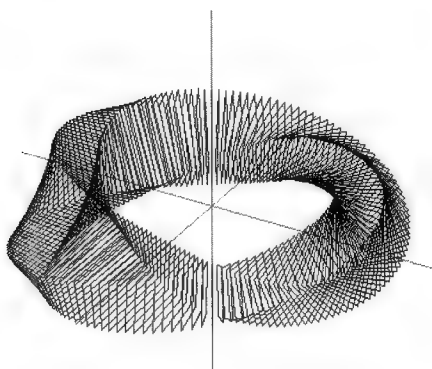
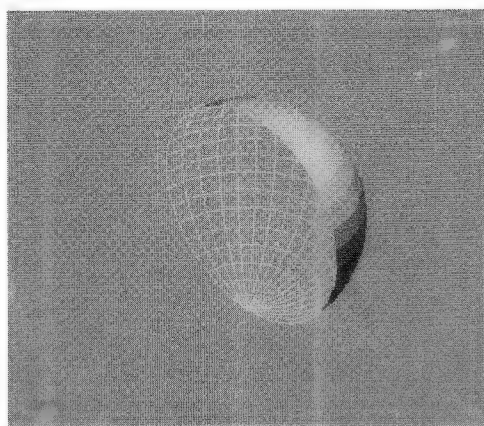
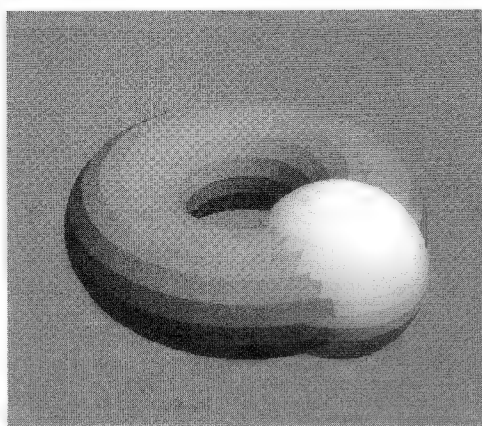
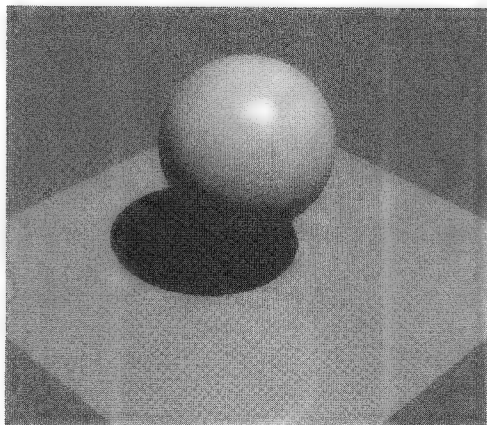
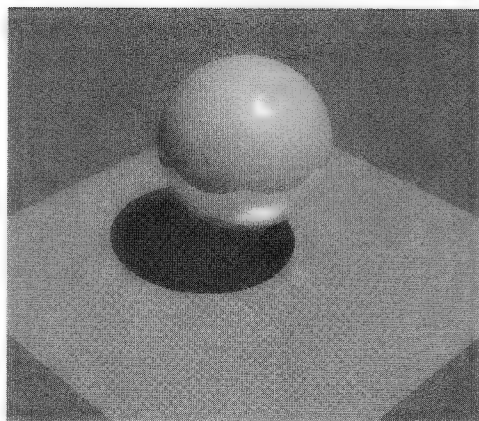
リスト PR5_2.cpp

```
 1: //      法線ベクトル法を利用した陰線（陰面）処理
 2: //      球体を描画する
 3:
 4: #include        "graph3.h"
 5:
 6: main()
 7: {
 8:     Line    a= circle_xz( 18, 0, 180);                  //  球体の素（半円）の生成
 9:     Surface f= revolve_z( 100*a, 36 );                  //  球体（ポリゴン集合）の生成
10:
11:     Win3D   w("EX5_2",WHITE);
12:     Point   V0=Point(500,500,500);                      //  視点の記述（座標：V0）
13:     w.setview(V0);
14:     w.color(RED);
15:
16:     //  球体（ポリゴン集合）の描画
17:     Vector  V,N;
18:     for(int i=0; i<f.n; i++){
19:         Polygon p=f.p[i];
20:         V=unit(p.p[0]-V0);                              //  視線ベクトル
21:         N=unit((p.p[0]-p.p[2])%(p.p[1]-p.p[3]));        //  法線ベクトルの計算（外積）
22:         if( N*V <=0 )w.line( p);                        //  ポリゴン枠の描画
23:     }
24:
25:     pause("EX5_2");
26: }
```

リスト PR7_1.cpp

```
 1: //  レイトレーシング法
 2: //  床と映り込みのある球体を表示する
 3:
 4: #include        "graph3.h"
 5: #include        "ray.h"
 6:
 7: main()
 8: {
 9:     Ball    Ba= Ball( Point(0,0,70),70);                    //  球体の記述
10:     Polygon Wp( Point( 200, 200,-50),Point(-200, 200,-50),
11:                 Point(-200,-200,-50),Point( 200,-200,-50) );
12:     Wall    Wa= Wall( Wp );                                 //  床の記述
13:     Color   Cb= RED;
14:     Color   Cw= 0.3*WHITE+0.4*BLUE;
15:
16:     Win3D   w("PR7_1");
17:     Point   V0=Point(500,500,500);                          //  視点の記述
18:     Point   L0=Point(0,400,1000);                           //  照明光の記述
19:     w.setview(V0);
20:
21:     for(int x=0; x<w.size_x; x++)
22:         for(int y=0; y<w.size_y; y++){
23:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);// 視線ベクトルの計算
24:             Ray     VR=Ray(V0,V);                           //  視線の記述
25:             float   s=1;                                    //  日向か日陰を表す
26:
27:             Point   Ps,Pr;                                  //  交点座標
28:             Vector  N,D;
29:             Color   Cs=0.25*CYAN,Cr;
30:             if(Ba.hit(VR,Ps,N)<INFINITY){                   //  視線が球体と交わるかの判定
31:                 Vector  L=unit(Ps-L0);                      //  照明光の方向ベクトル
32:                 Cs=shading(V,L,N,Cb,1);
33:             }
34:             else if(Wa.hit(VR,Ps,N)<INFINITY){              //  視線が床と交わるかの判定
35:                 Vector  L=unit(Ps-L0);                      //  照明光の方向ベクトル
36:                 Ray     LR=Ray(L0,L);                       //  照明光線の記述
37:                  if(Ba.hit(LR,Ps,D)<INFINITY)s=0;           //  照明光が床に当たるかかの判
定
38:                 Cs=shading(V,L,N,Cw,s);
39:                 Ray     RR=Ray(Ps, V-2*(V*N)*N);            //  反射視線の記述
40:                 if(Ba.hit(RR,Pr,N)<INFINITY){               //  視線が床と交わるかの判定
41:                     Vector  Lr=unit(Pr-L0);                 //  照明光の方向ベクトル
42:                     Ray     LR=Ray(L0, Lr);                 //  照明光線の記述
43:                     Cr=shading(RR.d,Lr,N,Cb,s);
44:                 }
45:             }
46:             w.color(Cs+0.3*Cr);
47:             w.point(x,y);
48:     }
49:     pause("PR7_1");
50: }
```

リスト PR8_1.cpp

```
 1: //  球体にバンプマッピングする
 2:
 3: #include        "graph1.h"
 4: #include        "graph3.h"
 5: #include        "ray.h"
 6:
 7: USERES("win.res");
 8: USELIB("Win.lib");
 9:
10: //  マッピング座標を返す
11: Point   mapping(Win2D& w, Vector& V, int
12:         float   a=0,        e=asin(V.z);               //  a:方位角 e:仰角
13:         int     wi=w.size_x,hi=w.size_y;
14:         if      (V.y>0)  a= acos(V.x/sqrt(V.x*V.x+V.y*V.y));
15:         else if(V.y<0)  a=-acos(V.x/sqrt(V.x*V.x+V.y*V.y));
16:         return  Point( (int)(dir*a*wi/PI/2)%wi, e*hi/PI );
17: }
18:
19: main()
20: {
21:     Ball    Ba= Ball( Point(0,0,0),100);               //  球体の記述
22:
23:     Win2D   tex("","..¥¥pic¥¥texture2.bmp");           //  テクスチャ画像用ウィンドウ
24:     Win2D   bum("","..¥¥pic¥¥bump2.bmp");              //  バンプ画像用ウィンドウ
25:
26:     Win3D   w("PR8_1", 0.25*CYAN);
27:     Point   V0=Point(500,200,200);                     //  視点の記述
28:     Point   L0=Point(500,-200,200);                    //  照明光の記述
29:     w.setview(V0);
30:
31:     for(int x=0; x<w.size_x; x++)
32:         for(int y=0; y<w.size_y; y++){
33:             Vector  V=unit(w.world(Point(x,y,-w.dv))-V0);//  視線ベクトルの計算
34:             Ray     VR=Ray(V0,V);                      //  視線の記述
35:
36:             Point   Ps,Pr;                             //  交点座標
37:             Vector  N;                                 //  法線ベクトル
38:             if(Ba.hit(VR,Ps,N)<INFINITY){              //  視線が球体と交わるかの判定
39:                 Vector  L=unit(Ps-L0);                 //  照明光の方向ベクトル
40:
41:                 Point   M= mapping(tex,N);             //  マッピング画像の座標
42:                 Color   c= tex.pixel(M);               //  マッピング画像の色
43:
44:                         M= mapping(bum,N);             //  マッピング画像の座標
45:
46:                 float   l00=5*(bum.pixel(M)).r;
47:                 float   l10=5*(bum.pixel(M+Point(1,0,0))).r;
48:                 float   l01=5*(bum.pixel(M+Point(0,1,0))).r;
49:
50:                 Vector  DX=Vector(1,0,l10-l00);
51:                 Vector  DY=Vector(0,1,l01-l00);
52:                 Vector  F =rot_z(90)*unit(DX%DY);
53:                         N=rotate(Point(0,0,1),F)*N;
54:
55:                 w.color( shading(V,L,N,c) );
56:                 w.point(x,y);
57:             }
58:     }
59:     pause("PR8_1");
60: }
```

実行結果（PR3_1.cpp）

実行結果（PR3_2.cpp）

実行結果（PR3_1.cpp）

実行結果（PR5_1.cpp）

実行結果（PR5_2.cpp）

実行結果（PR7_1.cpp）

実行結果（PR7_2.cpp）

# 巻末リスト

Win2D クラス(graph1.h)

```
 1: //      ２次元描画用のクラス定義
 2:
 3: #ifndef       __GRAPH1_H
 4: #define       __GRAPH1_H
 5:
 6: #include      "win.h"
 7: #include      "point.h"
 8: #include      "line.h"
 9: #include      "tmatrix.h"
10:
11: // ウィンドウ用のクラス
12: class  Win2D
13: {
14:     #define    HEIGHT_DEF  350
15:     #define    WIDTH_DEF   400
16:     public:
17:     int     size_x, size_y;                         // ウィンドウのサイズ
18:     float   org_x,  org_y;                          // 原点の位置（スクリーン座標）
19:     Win     w;
20:
21:     public:
22:     Win2D(char* wn="Win2D",int x=WIDTH_DEF, int y=HEIGHT_DEF, Color c=WHITE){
23:             w.open(wn,x,y,c);
24:             size_x=x;           size_y=y;
25:             org_x=size_x/2+.5;  org_y=size_y/2-.5;
26:     }
27:     Win2D(char* wn,char* fn){
28:             w.read(fn);
29:             size_x=w.wi;        size_y=w.hi;
30:             org_x=size_x/2+.5;  org_y=size_y/2-.5;
31:     }
32:     ~Win2D()                 {  w.close();  }
33:     void    origin(int x, int y){  org_x=x+.5; org_y=y-.5; }
34:     void    axis ();
35:     void    color(Color c)   {  w.color(c); }
36:     Color   pixel(Point p)   {  return w.pixel(org_x+p.x,org_y-p.y);    }
37:     void    point(Point p)   {  w.point(org_x+p.x,org_y-p.y);   }
38:     void    move (Point p)   {  w.move (org_x+p.x,org_y-p.y);   }
39:     void    line (Point p)   {  w.line (org_x+p.x,org_y-p.y);   }
40:     void    line (Point p0, Point p1){ move(p0);   line(p1);   }
41:     void    line (Point p[], int n);
42:     void    line (Line& l)   {  line(l.p, l.n); }
43:     void    clear()          {  w.clear();  }
44:     void    close()          {  w.close();  }
45:     #undef  HEIGHT_DEF
46:     #undef  WIDTH_DEF
47: };
48:
49: void    Win2D::line(Point p[], int n)
50: {       move(p[n-1]);   for(int i=0; i<n; i++)line(p[i]);   }
51:
52: void    Win2D::axis(){
53:         w.color(CYAN);
54:         line(Point(-org_x+10,0), Point(size_x-org_x-1-10,0));   //  X軸
55:         w.gprintf("X");
56:         line(Point(0,-size_y+org_y+10),Point(0,org_y-1-10));    //  Y軸
57:         w.gprintf("Y");
```

```
58:         w.color(BLACK);
59: }
60: #endif
```

Win3D クラス (graph2.h)

```
 1: //      3次元ウィンドウ用のクラス
 2:
 3: #ifndef         __GRAPH2_H
 4: #define         __GRAPH2_H
 5:
 6: #include        "win.h"
 7: #include        "point.h"
 8: #include        "line.h"
 9: #include        "tmatrix.h"
10:
11: //  3次元ウィンドウ用のクラス
12: class   Win3D{
13:         #define HEIGHT_DEF  350
14:         #define WIDTH_DEF   400
15:         public:
16:         Win     w;
17:         int     size_x, size_y;                     //  ウィンドウのサイズ
18:         Point   org;                                //  原点の位置（スクリーン座標）
19:         TMatrix t;                                  //  視点変換行列
20:         Point   v0;                                 //  視点の位置
21:
22:         Win3D(char* wn="Win3D",int x=WIDTH_DEF, int y=HEIGHT_DEF,Color c=WHITE){
23:                 w.open(wn,x,y,c);
24:                 size_x=x;           size_y=y;
25:                 org.x=size_x/2-0.5; org.y=size_y/2-1-.5;
26:                 setview(30,30,1000);
27:         }
28:         ~Win3D()     {   w.close();  }
29:         void    origin(int x, int y)        {   org=Point(x+.5,y-.5,0); }// 原点の位置設定
30:         void    setview(float a, float e, float d=1000){//視点方向の設定(d:原点からの距離)
31:                 t=perspect(d)*rot_x(e-90)*rot_z(-90-a); //  a:方位角  e:仰角  d:距離
32:                 v0=rot_z(90+a)*rot_x(90-e)*Point(0,0,1)*d;
33:         }
34:         void    setview(Point& p){                  //  視点位置の設定
35:                 Vector  wa=unit(p); v0=p;
36:                 if((wa.x==0)&&(wa.y==0)){   t=perspect(len(p)); return; }
37:                 float   wd=sqrt(wa.x*wa.x+wa.y*wa.y);
38:                 t=perspect(len(p))*rot_x(-wd,wa.z)*rot_z(-wa.x/wd,-wa.y/wd);
39:         }
40:         void    axis();                             //  座標軸の描画
41:         void    color(Color c)              {   w.color(c); }
42:         Point   screen(Point& p){                   //  スクリーン座標に変換する
43:                 Point   a=t*p;
44:                 float   s=t.t[3][0]*p.x+t.t[3][1]*p.y+t.t[3][2]*p.z+t.t[3][3];
45:                 return  Point( org.x+a.x/s, org.y-a.y/s, a.z );
46:         }
47:         void    point(Point& p)             {   Point a=screen(p); w.point(a.x,a.y);}
48:         void    move(Point& p)              {   Point a=screen(p); w.move(a.x,a.y); }
49:         void    line(Point& p)              {   Point a=screen(p); w.line(a.x,a.y); }
50:         void    line(Point& p0, Point& p1)  {   move(p0);   line(p1);   }
51:         void    line(Point p[], int n);
52:         void    line(Line& p)               {   line(p.p,p.n);  }
53:         void    clear()                     {   w.clear();  }
54:         void    close()                     {   w.close();  }
55: };
56:
57: void    Win3D::line(Point p[], int n){
58:         int*    q= new int[2*n];
59:         for(int i=0; i<n; i++){   Point a=screen(p[i]); q[2*i]=a.x; q[2*i+1]=a.y;   }
60:         w.line(q,n);
```

```
61:         delete(q);
62: }
63:
64: void    Win3D::axis(){
65:         w.color(CYAN);
66:         line( Line( Point(-size_x/2,0,0),Point(size_x/2,0,0) ) );   //  X軸
67:         w.gprintf("X");
68:         line( Line( Point(0,-size_x/2,0),Point(0,size_x/2,0) ) );   //  Y軸
69:         w.gprintf("Y");
70:         line( Line( Point(0,0,-size_y/2),Point(0,0,size_y/2) ) );   //  Z軸
71:         w.gprintf("Z");
72:         w.color(BLACK);
73: }
74: #endif
```

Point クラス（point.h）

```
 1: //  3次元（空間）の点のクラス
 2:
 3: #ifndef         __POINT_H
 4: #define         __POINT_H
 5:
 6: #include        <math.h>
 7:
 8: #define         PI  3.14159265359
 9: #define         PIR (PI/180)
10: #define         Vector  Point
11:
12: class   Point{
13:         public:
14:         float   x,y,z;                                  //各座標の値
15:
16:         Point()                             {   x=0;    y=0;    z=0;    }
17:         Point(float ix, float iy, float iz=1)   {   x=ix;   y=iy;   z=iz;       }
18: };
19:
20: //      演算しのオーバーロード
21: Point   operator+(Point& a,Point& b){   return Point(a.x+b.x, a.y+b.y, a.z+b.z);    }
22: Point   operator-(Point& a,Point& b){   return Point(a.x-b.x, a.y-b.y, a.z-b.z);    }
23: Point   operator-(Point& a)         {   return Point(-a.x, -a.y, -a.z); }
24: Point   operator*(float a, Point& b){   return Point(a*b.x, a*b.y, a*b.z);  }
25: Point   operator*(Point& a, float b){   return Point(a.x*b, a.y*b, a.z*b);  }
26: Point   operator/(Point& a, float b){   return Point(a.x/b, a.y/b, a.z/b);  }
27:
28: float   operator*(Point& a,Point& b){   return a.x*b.x+a.y*b.y+a.z*b.z; }   //内積
29: Point   operator%(Point& a,Point& b)                                        //外積
30: {       return Point(a.y*b.z-a.z*b.y,a.z*b.x-a.x*b.z, a.x*b.y-a.y*b.x); }
31: int     operator==(Point& a, Point& b){ return( a.x==b.x && a.y==b.y && a.z==b.z );}
32:
33: float   len(Point& a)
    {   return sqrt((double)a.x*a.x+(double)a.y*a.y+(double)a.z*a.z);   }   //大きさ（長さ）
34: Point   unit(Point& a)  {   return (len(a)==0.)?Point(1,0,0):a/len(a); }//大きさを１にする
35: Point   unit(float a, float e=0)                    //  方位角a,仰角eの単位ベクトルを返す
36: {       return Point( cos(a*PIR)*cos(e*PIR),sin(a*PIR)*cos(e*PIR),sin(e*PIR) );}
37:
38: Point   default_point;  //  省略判定用のダミー
39: #endif
```

Line クラス（line.h）

```
 1: //      3次元（空間）の線（複数線分）のクラス
 2: #ifndef         __LINE_H
 3: #define         __LINE_H
 4: #include        "point.h"
```

```
 5:
 6: class   Line{
 7:         public:
 8:         int n;                                              //  線分の数+1
 9:         Point*  p;                                          //  線分座標(Point)配列のポインタ
10:
11:         Line(int nn=0)   {   n=nn; if(nn>0) p= new Point[nn];    }
12:         #define DP      default_point
13:         Line(   Point& a0,    Point& a1,    Point& a2=DP, Point& a3=DP, Point& a4=DP,
14:                 Point& a5=DP, Point& a6=DP, Point& a7=DP, Point& a8=DP, Point& a9=DP );
15:         Line(Line& b)
            {   n=b.n; if(n>0) p= new Point[n]; for(int i=0; i<n; i++)p[i]=b.p[i]; }
16:         ~Line()          {   if(n>0) delete[] p; }
17:         Line    operator=(Line& a);
18: };
19:
20: Line::Line( Point& a0, Point& a1, Point& a2, Point& a3, Point& a4,
21:             Point& a5, Point& a6, Point& a7, Point& a8, Point& a9 ) {
22:         Point*  a[10]={ &a0, &a1, &a2, &a3, &a4, &a5, &a6, &a7, &a8, &a9 };
23:         for(n=0; (n<10)&&(a[n]!=&DP); n++);
24:         p= new Point[n];
25:         for(int i=0; i<n; i++)p[i]=*a[i];
26: }
27: #undef  DP
28:
29: Line    Line::operator=(Line& a){
30:         if(n>0) delete[] p;     n=a.n;
31:         if(n>0) p= new  Point[n];
32:         for(int i=0; i<n; i++)p[i]=a.p[i];
33:         return *this;
34: }
35: Line    operator*(float a, Line& b)
36: {       Line    c(b.n); for(int i=0; i<b.n; i++)c.p[i]=a*b.p[i]; return c;  }
37: float   len(Line& a){
38:         float   l=0.;
39:         for(int i=0; i<a.n-1; i++)  l+=len(a.p[i]-a.p[i+1]);
40:         return  l;
41: }
42:
43: //      折れ線（曲線に近似させた複数の線分）生成する関数
44: Line    circle_xz(int n, float b=0, float e=360){
45:         Line    l(n+1);
46:         float   t=PIR*b,    d=PIR*(e-b)/n;
47:         for(int i=0; i<n+1; i++,t+=d)l.p[i]=Point(sin(t),0,cos(t));
48:         return  l;
49: }
50: Line    circle_yz(int n, float b=0, float e=360){
51:         Line    l(n+1);
52:         float   t=PIR*b,    d=PIR*(e-b)/n;
53:         for(int i=0; i<n+1; i++,t+=d)l.p[i]=Point(0,cos(t),sin(t));
54:         return  l;
55: }
56: Line    circle_xy(int n, float b=0, float e=360){
57:         Line    l(n+1);
58:         float   t=PIR*b,    d=PIR*(e-b)/n;
59:         for(int i=0; i<n+1; i++,t+=d)l.p[i]=Point(cos(t),sin(t),0);
60:         return  l;
61: }
62: #endif
```

TMatrix クラス（tmatrix.h）

```
 1: //      ３次元（空間）の座標変換用行列
 2:
 3: #ifndef         __TMATRIX_H
 4: #define         __TMATRIX_H
```

```
 5:
 6: #include        "point.h"
 7: #include        "line.h"
 8: #include        "polygon.h"
 9:
10: //      座標変換用行列
11: class   TMatrix{
12:         public:
13:         float   t[4][4];                                //  変換行列
14:
15:         TMatrix()   {   init(); }
16:         TMatrix(float m00,      float m01,      float m02,
17:                 float m10,      float m11,      float m12,
18:                 float m20,      float m21,      float m22           ){
19:                 t[0][0]=m00;    t[0][1]=m01;    t[0][2]=m02;
20:                 t[1][0]=m10;    t[1][1]=m11;    t[1][2]=m12;
21:                 t[2][0]=m20;    t[2][1]=m21;    t[2][2]=m22;
22:         }
23:         TMatrix(float m00,      float m01,      float m02,      float m03,
24:                 float m10,      float m11,      float m12,      float m13,
25:                 float m20,      float m21,      float m22,      float m23,
26:                 float m30,      float m31,      float m32,      float m33   ){
27:                 t[0][0]=m00;    t[0][1]=m01;    t[0][2]=m02;    t[0][3]=m03;
28:                 t[1][0]=m10;    t[1][1]=m11;    t[1][2]=m12;    t[1][3]=m13;
29:                 t[2][0]=m20;    t[2][1]=m21;    t[2][2]=m22;    t[2][3]=m23;
30:                 t[3][0]=m30;    t[3][1]=m31;    t[3][2]=m32;    t[3][3]=m33;
31:         }
32:         void    init(){
33:                 for(int i=0; i<4; i++)  for(int j=0; j<4; j++)t[i][j]=0;
34:                 for( i=0; i<4; i++) t[i][i]=1;
35:         }
36: };
37:
38: Point   operator*(TMatrix& a, Point& b){
39:         return  Point(  a.t[0][0]*b.x+a.t[0][1]*b.y+a.t[0][2]*b.z+a.t[0][3],
40:                         a.t[1][0]*b.x+a.t[1][1]*b.y+a.t[1][2]*b.z+a.t[1][3],
41:                         a.t[2][0]*b.x+a.t[2][1]*b.y+a.t[2][2]*b.z+a.t[2][3] );
42: }
43: Line    operator*(TMatrix& a, Line& b)
44: {       Line c(b.n);    for(int i=0; i<c.n; i++)c.p[i]=a*b.p[i]; return c;  }
45: Polygon operator*(TMatrix& a, Polygon& b)
46: {       Polygon c(b.n); for(int i=0; i<c.n; i++)c.p[i]=a*b.p[i]; return c;  }
47: TMatrix operator*(TMatrix& a, TMatrix& b){
48:         TMatrix c;
49:         for(int i=0; i<4; i++)
50:             for(int j=0; j<4; j++){
51:                 c.t[i][j]=0;
52:                 for(int k=0; k<4; k++) c.t[i][j]+=a.t[i][k]*b.t[k][j];
53:             }
54:         return c;
55: }
56: TMatrix scale(float x, float y, float z)                //  倍率変換
57: {       TMatrix m;  m.t[0][0]=x; m.t[1][1]=y; m.t[2][2]=z;  return  m;  }
58: TMatrix scale(Point a)  {   return scale(a.x,a.y,a.z);  }
59: TMatrix scale(float a)  {   return scale(a,   a,   a);     }
60: TMatrix operator*(TMatrix& a, float b){ return a*scale(b,b,b);  }
61: TMatrix operator*(float a, TMatrix& b){ return scale(a,a,a)*b;  }
62:
63: TMatrix rot_x(float sn, float cs){                      //  X軸を中心に回転
64:         TMatrix m;  m.t[1][1]=cs;   m.t[1][2]=-sn;
65:                     m.t[2][1]=sn;   m.t[2][2]=cs;
66:         return  m;
67: }
68: TMatrix rot_y(float sn, float cs){                      //  Y軸を中心に回転
69:         TMatrix m;  m.t[0][0]=cs;   m.t[0][2]=sn;
70:                     m.t[2][0]=-sn;  m.t[2][2]=cs;
71:         return  m;
72: }
```

```
 73: TMatrix rot_z(float sn, float cs){                          //  Z軸を中心に回転
 74:         TMatrix m;  m.t[0][0]=cs;   m.t[0][1]=-sn;
 75:                     m.t[1][0]=sn;   m.t[1][1]=cs;
 76:         return  m;
 77: }
 78: TMatrix rot_x(float r)   {    return rot_x( sin(r*PIR), cos(r*PIR) ); }
 79: TMatrix rot_y(float r)   {    return rot_y( sin(r*PIR), cos(r*PIR) ); }
 80: TMatrix rot_z(float r)   {    return rot_z( sin(r*PIR), cos(r*PIR) ); }
 81: TMatrix rotate(Point& a, float sn, float cs){
 82:         Point   axy=unit( Point(a.x,a.y,0) );
 83:         Point   axz=rot_z(-axy.y, axy.x)*unit(a);
 84:         return  rot_z(axy.y, axy.x)*rot_y(-axz.z,axz.x)*rot_x(sn,cs)
 85:                     *rot_y(axz.z,axz.x)*rot_z(-axy.y, axy.x);
 86: }
 87: TMatrix rotate(float cs, float sn){                             //  回転変換
 88:         TMatrix t;
 89:         t.t[0][0]=t.t[1][1]=cs; t.t[1][0]=sn;   t.t[0][1]=-sn;
 90:         return  t;
 91: }
 92: TMatrix rotate(float r)          {    return rot_z(r);        }   //  回転変換
 93: TMatrix rotate(Point& a, float r){  return rotate(a, sin(r*PIR), cos(r*PIR) );  }
 94: TMatrix rotate(Point& a, Point& b){
 95:         float   sn=len(unit(a)%unit(b));
 96:         if(sn!=0)   return  rotate(unit(a%b),sn,unit(a)*unit(b));
 97:         return  TMatrix();
 98: }
 99: TMatrix rotate(Point n)  {                                      //  ベクトル方向に回転
100:         Point   ss= (Point(1,0,0)%unit(n) );
101:         if( len(ss)==0. ) ss=Point(0,0,1);
102:         Point   s=unit( ss );
103:         TMatrix t1=rot_x(-s.y,s.z), t3=rot_x(s.y,s.z);
104:         Point   m=unit(t1*n);
105:         TMatrix t2=rot_z(m.y,m.x);
106:         return  t3*t2*t1;
107: }
108: TMatrix move(float x, float y, float z)                         //  平行移動
109: {       TMatrix m;  m.t[0][3]=x;    m.t[1][3]=y;    m.t[2][3]=z;    return  m;  }
110: TMatrix move(Point a){  return move(a.x,a.y,a.z);   }   //  平行移動
111: //      透視変換 (遠近感)
112: TMatrix perspect(float d){  TMatrix m; m.t[2][3]=-d;m.t[3][2]=-1/d; return  m;  }
113: TMatrix trans(TMatrix& a){                                      //  転置行列
114:         TMatrix c;
115:         for(int i=0; i<4; i++)
116:             for(int j=0; j<4; j++)  c.t[i][j]=a.t[j][i];
117:         return c;
118: }
119: TMatrix inverse(TMatrix& a){                                    //  逆変換行列
120:         float   d=0;
121:         for(int i=0; i<3; i++)
122:             d=d+a.t[0][i]*a.t[1][(i+1)%3]*a.t[2][(i+2)%3]
123:                 -a.t[0][i]*a.t[1][(i+2)%3]*a.t[2][(i+1)%3];
124:         TMatrix b;
125:         if(d!=0)
126:         for(  i=0; i<3; i++)                                    //  拡大・回転
127:             for(int j=0; j<3; j++)
128:                 b.t[i][j]=(a.t[(j+1)%3][(i+1)%3]*a.t[(j+2)%3][(i+2)%3]
129:                         -a.t[(j+1)%3][(i+2)%3]*a.t[(j+2)%3][(i+1)%3])/d;
130:         for(  i=0; i<3; i++)b.t[i][3]=-a.t[i][3];            //  移動
131:         return b;
132: }
133: #endif
```

Polygon クラス (polygon.h)

```
  1: //      3次元 (空間) の多角形のクラス
  2:
```

```
 3: #ifndef        __POLYGON_H
 4: #define        __POLYGON_H
 5:
 6: #include       "point.h"
 7:
 8: // 多角形のクラス
 9: class  Polygon{
10:        public:
11:        int n;                                           // 線分の数＋1
12:        Point* p;                                        // 頂点座標(Point)配列のポインタ
13:
14:        Polygon(int nn=0)      {  if(nn>0) p= new Point[nn];n=nn;}
15:        Polygon(Point a[], int nn);
16:        #define DP  default_point
17:        Polygon(Point& a0,   Point& a1,   Point& a2=DP, Point& a3=DP, Point& a4=DP,
18:               Point& a5=DP, Point& a6=DP, Point& a7=DP, Point& a8=DP, Point& a9=DP );
19:        Polygon::Polygon(Polygon& b)
20:        {      n=b.n;  if(n>0) p= new Point[n]; for(int i=0; i<n; i++)p[i]=b.p[i]; }
21:        Polygon(Line& a);
22:        ~Polygon()          {  if(n>0) delete[] p; }
23:        Polygon operator=(Polygon& a);
24: };
25:
26: Polygon::Polygon(Point a[], int nn) {
27:        if(nn>0) p= new Point[nn+1];   n=nn+1;
28:        for(int i=0; i<n-1; i++)p[i]=a[i]; p[n-1]=a[0];
29: }
30:
31: Polygon::Polygon(Point& a0, Point& a1, Point& a2, Point& a3, Point& a4,
32:               Point& a5, Point& a6, Point& a7, Point& a8, Point& a9 ){
33:        Point*  a[10]={ &a0, &a1, &a2, &a3, &a4, &a5, &a6, &a7, &a8, &a9 };
34:        for(n=0; (n<10)&&(a[n]!=&DP); n++);
35:        p= new Point[n+1];
36:        for(int i=0; i<n; i++)p[i]=*a[i];
37:        p[n]=*a[0]; n++;
38: }
39:        #undef  DP
40:
41: Polygon::Polygon(Line& a){
42:        if(a.p[0]==a.p[a.n-1])
43:        {  n=a.n;  if(n>0)p= new Point[n]; for(int i=0; i<n; i++)p[i]=a.p[i];  }
44:        else{
45:           n=a.n+1;if(n>0) p= new Point[n];
46:           for(int i=0; i<n-1; i++)p[i]=a.p[i];
47:           p[n-1]=a.p[0];
48:        }
49: }
50:
51: Polygon Polygon::operator=(Polygon& a){
52:        if(n>0) delete[] p;
53:        n=a.n;  if(n>0) p= new  Point[n];
54:        for(int i=0; i<n; i++)p[i]=a.p[i];
55:        return *this;
56: }
57: #endif
```

Surface クラス (surface.h)

```
1: //      サーフェイスモデルを記述するためのクラス
2: #ifndef        __SURFACE_H
3: #define        __SURFACE_H
4: #include       "point.h"
5: #include       "line.h"
6: #include       "polygon.h"
7: #include       "tmatrix.h"
8:
```

```
 9: class   Surface{
10:         public:
11:         int n;                                          //  Polygon の数
12:         Polygon* p;                                     //  Polygon 配列のポインタ
13:         Surface (int a=0)    {    n=a; if(n>0)p= new Polygon[n];  }
14:         Surface (Polygon a) {    n=1; p= new Polygon[n]; p[0]=a; }
15:         Surface (Surface& a)
16:         {       n=a.n;  if(n>0) p= new Polygon[n];  for(int i=0; i<n; i++)p[i]=a.p[i];  }
17:         ~Surface()          {   if(n>0) delete[] p;     }
18:         Surface operator=(Surface& a);
19: };
20:
21: Surface Surface::operator=(Surface& a)  {
22:         if(n>0) delete[] p; n=a.n;
23:         if(n>0) p= new  Polygon[n];
24:         for(int i=0; i<n; i++)p[i]=a.p[i];
25:         return *this;
26: }
27: Surface operator*(TMatrix& a, Surface& b)
28: {       Surface c(b.n); for(int i=0; i<c.n; i++)c.p[i]=a*b.p[i]; return c; }
29:
30: //  線分 a を Z 軸回転てポリゴンモデルを作る
31: Surface revolve_z(Line& a, int n, float b=0, float e=360)   {
32:         Surface f( (a.n-1)*(n) );
33:         TMatrix m0,m1;
34:         for(int j=0, t=0; j<n; j++) {
35:             m0=rot_z(b+j*(e-b)/n);
36:             m1=rot_z(b+(j+1)*(e-b)/n);
37:             for(int i=0; i<a.n-1; i++, t++)
38:             f.p[t]=Polygon( m0*a.p[i],m0*a.p[i+1],m1*a.p[i+1],m1*a.p[i] );
39:         }
40:         return f;
41: }
42:
43: TMatrix rot_xz(Vector& a){      //  vector(0,1,0)
44:         Vector  ua=unit(a), aa=scale(1,1,0)*ua;
45:         if(len(aa)==0)return rot_x(90);
46:         TMatrix mx=rot_x(Vector(0,0,1)*ua,len(aa));
47:         TMatrix mz=rot_z(-aa.x/len(aa),aa.y/len(aa));
48:         return mz*mx;
49: }
50: //  線分 a を線分 b に沿ってポリゴンモデルを作る
51: Surface sweep_xz(Line& a, Line& b)  {
52:         Point*  p=new Point[(a.n)*(b.n)];
53:         Line*   f=new Line[b.n];
54:         for( int j=0; j<b.n; j++)f[j].p=&p[j*a.n];
55:         if(len(b.p[0]-b.p[b.n-1])>0.1){
56:             f[0]=     move(b.p[0])*rot_xz(b.p[1]-b.p[0])*a;
57:             f[b.n-1]=move(b.p[b.n-1])*rot_xz(b.p[b.n-1]-b.p[b.n-2])*a;
58:         }   else    {
59:             f[0]=     move(b.p[0])*rot_xz(b.p[1]-b.p[b.n-4])*a;
60:             f[b.n-1]=move(b.p[0])*rot_xz(b.p[3]-b.p[b.n-2])*a;
61:         }
62:         for( int i=1,t; i<b.n-1; i++)
63:             f[i]=move(b.p[i])*rot_xz(b.p[i+1]-b.p[i-1])*a;
64:         Surface s( (a.n-1)*(b.n-1) );
65:         for( i=0,t=0; i<b.n-1; i++)
66:             for( j=0; j<a.n-1; j++,t++)
67:                 s.p[t]=Polygon( f[i].p[j],f[i].p[j+1],f[i+1].p[j+1],f[i+1].p[j]);
68:         return s;
69: }
70: #endif
```

Win3D クラス (graph3.h)

```
1: //      3次元ウィンドウ用のクラス
```

```
 2:
 3: #ifndef         __GRAPH3_H
 4: #define         __GRAPH3_H
 5:
 6: #include        "win.h"
 7: #include        "point.h"
 8: #include        "line.h"
 9: #include        "polygon.h"
10: #include        "tmatrix.h"
11: #include        "surface.h"
12:
13: //  ３次元ウィンドウ用のクラス
14: class   Win3D{
15:         #define HEIGHT_DEF  350
16:         #define WIDTH_DEF   400
17:         public:
18:         Win     w;
19:         int     size_x, size_y;                         //  ウィンドウのサイズ
20:         Point   org;                                    //  原点の位置（スクリーン座標）
21:         TMatrix t,it;                                   //  視点変換行列,t の逆行列
22:         Point   v0;                                     //  視点の位置
23:         float   dv;
24:
25:         Win3D(char* wn="Win3D", int x=WIDTH_DEF, int y=HEIGHT_DEF,Color c=BLACK){
26:                 w.open(wn,x,y,c);
27:                 size_x=x;               size_y=y;
28:                 org.x=size_x/2-.5;  org.y=size_y/2-1-.5;
29:                 setview(30,30,1000);color(WHITE);
30:         }
31:         Win3D(char* wn, Color c, int x=WIDTH_DEF, int y=HEIGHT_DEF){
32:                 w.open(wn,x,y,c);
33:                 size_x=x;               size_y=y;
34:                 org.x=size_x/2-.5;  org.y=size_y/2-1-.5;
35:                 setview(30,30,1000);color(WHITE);
36:         }
37:         ~Win3D()                        {   w.close();   }
38:         void    origin(int x, int y)    {   org=Point(x+.5,y-.5,0); }//  原点の位置設定
39:         void    setview(float a, float e, float d=100000){      //視点方向の設定(d:距離)
40:                 dv=d;
41:                 t=perspect(d)*rot_x(e-90)*rot_z(-90-a);
42:                 it=rot_z(90+a)*rot_x(90-e);
43:                 v0=it*Point(0,0,1)*d;
44:         }
45:         void    setview(Point& p){                      //  視点位置の設定
46:                 dv=len(p);
47:                 Vector  wa=unit(p); v0=p;
48:                 if((wa.x==0)&&(wa.y==0))    {   t=perspect(len(p)); return; }
49:                 float   wd=sqrt(wa.x*wa.x+wa.y*wa.y);
50:                 t=perspect(len(p))*rot_x(-wd,wa.z)*rot_z(-wa.x/wd,-wa.y/wd);
51:                 it=rot_z(wa.x/wd,-wa.y/wd)*rot_x(wd,wa.z);
52:         }
53:         void    axis();                                 //  座標軸の描画
54:         void    color(Color c)          {   w.color(c); }
55:         Point   screen(Point& p){                       //  スクリーン座標に変換する
56:                 Point   a=t*p;
57:                 float   s=t.t[3][0]*p.x+t.t[3][1]*p.y+t.t[3][2]*p.z+t.t[3][3];
58:                 return  Point( org.x+a.x/s, org.y-a.y/s, a.z );
59:         }
60:         Point   world(Point& p){                //  視野(view)座標を world 座標に変換する
61:                 float   sz=-p.z/dv;
62:                 return  it*Point((p.x-org.x)*sz,(-p.y+org.y)*sz,p.z+dv);
63:         }
64:         void    point2(Point& p)        {   w.point(p.x,p.y);   }       //  screen 座標
65:         void    point(int x, int y)     {   w.point(x,y);   }
66:         void    line2(Point& p0,Point& p1)  {   w.move(p0.x,p0.y);  w.line(p1.x,p1.y);  }
67:         void    point(Point& p)         {   point2( screen(p) );    }
68:         void    line (Point& p0, Point& p1) {   line2( screen(p0),screen(p1) ); }
69:         void    line (Point p[], int n);
```

```
 70:         void    line (Line& p)          {   line(p.p,p.n);  }
 71:         void    line (Polygon& p)       {   line(p.p,p.n);  }
 72:         void    paint(Polygon& p);
 73:         void    line(Surface& p);
 74:         //void  read(char* fn)          {   w.read(fn); size_x=w.wi; size_y=w.hi;   }
 75:         //void  write(char* fn)         {   w.write(fn);    }
 76:         void    clear()                 {   w.clear();  }
 77:         void    close()                 {   w.close();  }
 78: };
 79:
 80: void    Win3D::line(Point p[], int n){
 81:         int*    q= new int[2*n];
 82:         for(int i=0; i<n; i++){   Point a=screen(p[i]); q[2*i]=a.x; q[2*i+1]=a.y;   }
 83:         w.line(q,n);
 84:         delete(q);
 85: }
 86:
 87: void    Win3D::paint(Polygon& p){
 88:         int*    q= new int[2*p.n];
 89:         Point   a;
 90:         for(int i=0; i<p.n; i++){   a=screen(p.p[i]);   q[2*i]=a.x; q[2*i+1]=a.y;   }
 91:         w.paint(q,p.n);
 92:         delete(q);
 93: }
 94:
 95: //      Surface (ポリゴン集合)の描画
 96: void    Win3D::line(Surface& f){    for(int i=0; i<f.n; i++) line( f.p[i] );    }
 97:
 98: void    Win3D::axis(){
 99:         w.color(CYAN);
100:         line( Line( Point(-size_x/2,0,0),Point(size_x/2,0,0) ) );    //  X軸
101:         line( Line( Point(0,-size_x/2,0),Point(0,size_x/2,0) ) );    //  Y軸
102:         line( Line( Point(0,0,-size_y/2),Point(0,0,size_y/2) ) );    //  Z軸
103:         w.color(BLACK);
104: }
105: #endif
```

Ray クラス (ray.h)

```
 1: //  光線のクラス、球体クラス、壁クラス (レイトレーシング法用)
 2: #ifndef         __RAY_H
 3: #define         __RAY_H
 4: #define         INFINITY    1e8                     //  無限遠方
 5: #define         DEL         0.5                     //  計算誤差
 6:
 7: //  視線、光線のクラス
 8: class   Ray{
 9:         public:
10:         Point   o;                                  //  始点座標
11:         Vector  d;                                  //  方向ベクトル
12:         Ray     (Point& or, Vector& di){    o=or;   d=di;   }
13: };
14: //  球体のクラス
15: class   Ball{
16:         public:
17:         Point   o;                                  //  中心座標
18:         float   r;                                  //  半径
19:         Ball(Point& or=Point(0,0,0), float ra=0){   o=or; r=ra; }
20:         float   hit(Ray& VR, Point& P, Vector& N);  //  交点座標を返す
21:         float   hit_bv(Ray& VR);                    //  VB
22: };
23: float   Ball::hit(Ray& VR, Point& P, Vector& N){    //  交点座標を返す
24:         float   ds=VR.d*(VR.o-o);
25:         float   d2=ds*ds-(VR.o-o)*(VR.o-o)+r*r;     //  判別式
26:         if( d2 < 0 )    return INFINITY;
27:         float   t=-ds-sqrt(d2);                     //  距離
```

```
28:         if(t<DEL) t=-ds+sqrt(d2);
29:         P=VR.o+t*VR.d;  N=unit(P-o);                    //  交点座標,法線ベクトル
30:         return t;
31: }
32: float   Ball::hit_bv(Ray& VR){                          //  BV(バウンディングボリューム)
33:         float   ds=VR.d*(VR.o-o);                       //          の交差判定
34:         float   d2=ds*ds-(VR.o-o)*(VR.o-o)+r*r;         //  判別式
35:         if( d2 < 0 )    return INFINITY;
36:         return  -ds-sqrt(d2);                           //  距離
37: }
38: //  壁のクラス
39: class   Wall:public Polygon{
40:         public:
41:         Vector  nv;                                     //  法線ベクトル
42:         Wall()  {}
43:         Wall(Polygon& pi):Polygon(pi){  nv=unit((p[0]-p[2])%(p[1]-p[3]));   }
44:         float   hit(Ray& VR, Point& P, Vector& N);      //  交点座標を返す
45: };
46: float   Wall::hit(Ray& VR, Point& P, Vector& N){        //  交点座標を返す
47:         float   t=nv*(p[1]-VR.o)/(nv*VR.d);             //  距離
48:         if(t<=0.)   return INFINITY;
49:         P=VR.o+t*VR.d;  N=nv;                           //  交点,法線
50:         for(int i=0; i<n-1; i++)                        //  交点座標Pがポリゴン内かの判定
51:             if(N*((p[i+1]-p[i])%(P-p[i]))<0)return INFINITY;
52:         return t;
53: }
54: //  反射光を計算する    V:視線ベクトル  L:照明方向ベクトル  N:法線ベクトル  s:照明光強度
55: Color   shading(Vector& V, Vector& L, Vector& N, Color& c,float s=1){
56:         float   kd=0.7, ks=0.7, ke=0.3;                 //拡散反射係数,鏡面反射係数,環境光
57:         Vector  R=L-2*(L*N)*N;                          //  反射方向ベクトル
58:         return  kd*max(-N*L,0)*s*c+ks*pow(max(-R*V,0),20)*s*WHITE+ke*c;
59: }
60: #endif
```

Attri クラス (atrri.h)

```
 1: #ifndef         __ATTRI_H
 2: #define         __ATTRI_H
 3:
 4: //  光学的性質(Attri)を記述
 5: class   Attri
 6: {
 7:     public:
 8:     Color   c,*tx;                              //  色,模様
 9:     float   kd,ks,ke;                           //  拡散反射率,鏡面反射率,環境光係数
10:     float   kr,kn,kt;                           //  反射率,屈折率,透過率
11:     int     wi,hi,*rn;                          //  ウィンドウサイズ
12:     Attri   (Color ic=WHITE, float id=0.7, float is=0.8, float ie=0.2,
13:                             float ir=0, float in=0 ,float it=1){
14:             kd=id;      ks=is;      ke=ie;      kr=ir;      kn=in;  kt=it;
15:             c=ic;       wi=0;       tx=NULL;    rn=new int; *rn=1;
16:     }
17:     Attri   (char* fn, float id=0.7, float is=0.8, float ie=0.2,
18:                             float ir=0, float in=0 ,float it=1){
19:             kd=id;      ks=is;      ke=ie;      kr=ir;      kn=in;  kt=it;
20:             Win w;      w.read(fn); wi=w.wi;    hi=w.hi;
21:             c=WHITE;    tx=new Color[wi*hi];    rn=new int; *rn=1;
22:             for(int i=0; i<wi*hi; i++)  tx[i]=w.pixel(i%wi,i/wi);
23:     }
24:     Attri   (Attri& a){
25:             kd=a.kd;    ks=a.ks;    ke=a.ke;    kr=a.kr;    kn=a.kn;    kt=a.kt;
26:             c=a.c;      wi=a.wi;    hi=a.hi;    tx=a.tx;
27:             rn=a.rn;    if(rn!=NULL)(*rn)++;
28:     }
29:     Attri   operator=(Attri& a){
30:             kd=a.kd;    ks=a.ks;    ke=a.ke;    kr=a.kr;    kn=a.kn;    kt=a.kt;
```

```
31:             c=a.c;      wi=a.wi;    hi=a.hi;    tx=a.tx;
32:             rn=a.rn;    if(rn!=NULL)(*rn)++;
33:             return *this;
34:     }
35:     ~Attri()    {   (*rn)--; if( rn==NULL && tx==NULL )delete[] tx; }
36: };
37:
38: //  反射光を計算する(フォンモデル)
39: //  V:視線ベクトル  L:照明光の方向ベクトル  N:表面の法線ベクトル
40: Color   shading(Vector& V, Vector& L, Vector& N, Attri& A, float s=1){
41:         Vector  R=L-2*(L*N)*N;                          //  反射方向ベクトル
42:         return  (A.kd*max(-N*L,0)*s+A.ke)*A.c+A.ks*pow(max(-R*V,0),20)*s*WHITE;
43: }
44:
45: #endif
```

Csgm クラス (csgm.h)

```
 1: //  ＣＳＧモデル
 2:
 3: #ifndef         __CSGM_H
 4: #define         __CSGM_H
 5:
 6: #include        "ray.h"
 7:
 8: //  ２次曲面のクラス
 9: class   Quad{
10:         public:
11:         int     type;                                   //  次数
12:         TMatrix A;                                      //  ２次の係数
13:         Vector  B;                                      //  １次の係数
14:         float   C;                                      //  ０次の係数
15:         Quad()      {   type=0; C=0;    }
16:         Quad(Vector& s2,Vector& s1,float s0){
17:             A=scale(s2);    B=s1;   C=s0;
18:             if      ((s2==Point(0,0,0))==0) type=2;
19:             else if((s1==Point(0,0,0))==0)  type=1;
20:             else    type=0;
21:         }
22:         int     inner(Point& P)     {
23:                 if(type==1) if(B*P+C<=0)return 1; else return 0;
24:                 else if( (A*P)*P+B*P+C<=0)return 1; return 0;
25:         }
26:         float   hit(Ray& VR, Point& P, Vector& N);
27: };
28:
29: //  視線が２次曲面と交わるかのチェック (交点座標Ｐを返す)
30: float   Quad::hit(Ray& VR, Point& P, Vector& N){
31:         Point   V0=VR.o;
32:         Vector  V= VR.d;
33:         float   t= INFINITY,    ad=0;
34:         if(type==0) return t;
35:         if(type==1) {   t=-(B*V0+C)/(B*V);  }
36:         else{
37:             float   a=(A*V)*V,  b=(2*A*V0+B)*V,  c=(A*V0+B)*V0+C;
38:             float   d=b*b-4*a*c;                        //  判別式
39:             if( d<0 )   return INFINITY;
40:             ad=sqrt(d)/fabs(2*a);
41:             t=-b/(2*a)-ad;                              //  距離
42:             if( t<DEL )t=t+2*ad;
43:         }
44:         if( t<DEL ) return INFINITY;
45:         P=VR.o+t*VR.d;
46:         N=unit(2*A*P+B);
47:         return t;
48: }
```

```
 49:
 50: Quad      operator*(TMatrix& a, Quad& b){                  //  ２次曲面の移動
 51:           Quad     c=b;
 52:           TMatrix r=inverse(a);                            //  逆行列の計算
 53:           Vector   o=-(a*Point(0,0,0));
 54:           r.t[0][3]=r.t[1][3]=r.t[2][3]=0;
 55:           c.A=trans(r)*b.A*r; c.B=trans(r)*b.B+2*c.A*o;
 56:           c.C=b.C+(c.B-c.A*o)*o;
 57:           return c;
 58: }
 59: Quad      operator-(Quad& b)                               //  内外を逆に
 60: {         Quad     c=b; c.A=-1*b.A; c.B=-b.B; c.C=-b.C; return c;  }
 61:
 62: #define      AND      1
 63: #define      OR       2
 64: #define      MINUS    3
 65:
 66: //  CSG 用のクラス
 67: class     Csgm{
 68:           public:
 69:           Csgm     *c1,*c2;                                //  Csgm へのポインタ
 70:           Quad     *h1;                                    //  ２次曲面へのポインタ
 71:           int      op;                                     //  演算の指定
 72:           Ball     BV;                                     //  バウンディングボリューム
 73:           Csgm     ()  {   op=0;   c1=c2=NULL; h1=NULL; }
 74:           Csgm     ( Vector& s2,Vector& s1,float s0)
 75:           {        c1=c2=NULL; h1=new Quad;    *h1=Quad(s2,s1,s0); op=NULL;     }
 76:           Csgm     ( Csgm& a, Csgm& b, int md=0);
 77:           Csgm     ( Csgm& a);
 78:           Csgm     ( Csgm& a, Ball& bv);
 79:           Csgm     operator=( Csgm& a );
 80:           float    hit(Ray& VR, Point& P, Vector& N);
 81:           int      inner(Point& P) {                       //  内部の判定
 82:                    if(op==AND )return c1->inner(P)*c2->inner(P);
 83:                    if(op==OR  )return c1->inner(P)|c2->inner(P);
 84:                    return h1->inner(P);
 85:           }
 86:           int      outer(Point& P) {   if(inner(P)) return 0; return 1;    }  //  内部の判定
 87:           ~Csgm() {   if(c1!=NULL)delete c1; if(c2!=NULL)delete c2; if(h1!=NULL)delete h1;    }
 88: };
 89:
 90: Csgm::Csgm( Csgm& a){
 91:           BV=a.BV;
 92:           c1=c2=NULL; h1=NULL;
 93:           if(a.op!=NULL){
 94:               c1=new Csgm;    c2=new Csgm;
 95:               *c1=*a.c1;      *c2=*a.c2;
 96:           } else  { h1=new Quad;  *h1=*a.h1;  }
 97:           op=a.op;
 98: }
 99: Csgm::Csgm( Csgm& a, Ball& bv ){
100:           BV=bv;
101:           c1=c2=NULL; h1=NULL;
102:           if(a.op!=NULL){
103:               c1=new Csgm;    c2=new Csgm;
104:               *c1=*a.c1;      *c2=*a.c2;
105:           } else  { h1=new Quad;  *h1=*a.h1;  }
106:           op=a.op;
107: }
108: Csgm      Csgm::operator=( Csgm& a ){
109:           BV=a.BV;
110:           c1=c2=NULL; h1=NULL;
111:           if(a.op!=NULL){
112:               c1=new Csgm;    c2=new Csgm;
113:               *c1=*a.c1;      *c2=*a.c2;
114:           } else  { h1=new Quad;  *h1=*a.h1;  }
115:           op=a.op;
116:           return *this;
```

```
117: }
118:
119: Csgm    operator*(TMatrix& a, Csgm& b){          // Csgmの移動
120:         Csgm    c=b;
121:         if( b.op==NULL ){   *c.h1=a*(*b.h1);     return c;   }
122:         *c.c1=a*(*b.c1);    *c.c2=a*(*b.c2);     return c;
123: }
124: Csgm    operator-(Csgm& b){                      // 内外を逆に
125:         Csgm    c=b;
126:         if( b.op==NULL ){   *c.h1=-(*b.h1); return c;   }
127:         if( b.op==AND ) c.op=OR;
128:         else if( b.op==OR )  c.op=AND;
129:         *c.c1=-(*b.c1); *c.c2=-(*b.c2); return c;
130: }
131:
132: Csgm::Csgm( Csgm& a, Csgm& b, int md){           // 演算の指定
133:         BV=Ball(Point(0,0,0),-1);
134:         c1=c2=NULL; h1=NULL;
135:         c1=new Csgm;    c2=new Csgm;
136:         if(md==MINUS){  *c2=-b; op=AND; }
137:         else        {   *c2=b;  op=md;  }
138:         *c1=a;
139: }
140:
141: //  Csgm間の演算の定義
142: Csgm    operator+(Csgm& a, Csgm& b){    return Csgm( a, b, OR );    }
143: Csgm    operator-(Csgm& a, Csgm& b){    return Csgm( a, b, MINUS ); }
144: Csgm    operator*(Csgm& a, Csgm& b){    return Csgm( a, b, AND );   }
145:
146: //  視線が球体と交わるかのチェック（交点座標Pを返す）
147: float   Csgm::hit(Ray& VR, Point& P, Vector& N){
148:         if( (BV.r>0) && (BV.hit(VR,P,N)>=INFINITY) )return INFINITY;
149:         Point   P1,P2,N1,N2;
150:         float   t1,t2;
151:         if( op==AND ){
152:             Ray V=VR;
153:             for( int i=0; i<10; i++){
154:                 t1=c1->hit(V,P1,N1);
155:                 t2=c2->hit(V,P2,N2);
156:                 if( t1>=INFINITY && t2>=INFINITY )return INFINITY;
157:                 if( fabs(t1-t2)<DEL ){
158:                     Point   PP=V.o+(max(t1,t2)+DEL)*V.d;
159:                     if( c1->inner(PP)&&c2->inner(PP) )
160:                             {   P=V.o+max(t1,t2)*V.d;   N=N1;   return len(VR.o-P); }
161:                     else return INFINITY;
162:                 }
163:                 if( t1<t2 )
164:                     if(c2->inner(P1)){  N=N1;   P=P1;   return len(VR.o-P1);    }
165:                     else            {   V.o=P1; }
166:                 else if(c1->inner(P2)){ N=N2;   P=P2;   return len(VR.o-P2);    }
167:                     else            {   V.o=P2; }
168:             }
169:             return INFINITY;
170:         } else if( op==OR ){
171:             Ray V=VR;
172:             for( int i=0; i<10; i++){
173:                 t1=c1->hit(V,P1,N1);
174:                 t2=c2->hit(V,P2,N2);
175:                 if( t1>=INFINITY && t2>=INFINITY )  return INFINITY;
176:                 if( t1<t2 )
177:                     if(c2->outer(P1)){  P=P1;   N=N1;   return len(VR.o-P1);    }
178:                     else            {   V.o=P1; }
179:                 else if(c1->outer(P2)){ P=P2;   N=N2;   return len(VR.o-P2);    }
180:                     else            {   V.o=P2; }
181:             }
182:         } else if( op==NULL ){  return h1->hit(VR,P,N);     }
183:         return INFINITY;
184: }
185: Csgm    ball(Point& s){
```

```
186:         Point   s2((s.x==0)?0:1/(s.x*s.x), (s.y==0)?0:1/(s.y*s.y), (s.z==0)?0:1/(s.z*s.z));
187:         return Csgm( s2, Point(0,0,0), -1 );
188: }
189: Csgm     ball( float r){ return ball( Point(r,r,r) );     }
190: Csgm     cylinder(Point& s){ return ball(s); }
191: Csgm     plane(Point& a, float b=0){ return Csgm(Point(0,0,0),a,b);  }
192: Csgm     plane_xy()  {   return Csgm(Point(0,0,0),Point(0,0,1),0);   }
193: Csgm     plane_yz()  {   return Csgm(Point(0,0,0),Point(1,0,0),0);   }
194: Csgm     plane_xz()  {   return Csgm(Point(0,0,0),Point(0,1,0),0);   }
195: Csgm     board_xy(float b=10){
196:          return  Csgm(Point(0,0,0),Point(0,0,-1),-b/2)
197:                  *Csgm(Point(0,0,0),Point(0,0,1),-b/2);
198: }
199: #endif
```

Scene クラス (scene.h)

```
 1: //  場面を記録する,及び描画にする(影,多重反射,屈折の処理も行う)
 2: //  レイトレーシング
 3: #ifndef          __SCENE_H
 4: #define          __SCENE_H
 5:
 6: #include         "ray.h"
 7: #include         "attri.h"
 8: #include         "csgm.h"
 9:
10: //  複数の物体(壁、球体、CSG モデル、自由曲面)を記憶するクラス
11: #define     MAX_OBJ     1024                                  //  記憶する物体の最大数
12: #define     WALL        0x10
13: #define     BALL        0x20
14: #define     CSGM        0x30
15: #define     FREE        0x40
16: #define     BVOL        0xF0
17: class   Scene{
18:         public:
19:         void*   obj[MAX_OBJ];                         //  物体へのポインタ
20:         int     type[MAX_OBJ],n;                      //  物体の種類,物体の個数
21:         int     ele[MAX_OBJ];                         //  個数
22:         Attri   att[MAX_OBJ];                         //  物体の光学的特性
23:         Color   bgc;                                  //  背景色
24:         int     md;                                   //  表示の有無
25:
26:         Scene(Color b=BLACK, int m=1)
27:                 {   n=0;    md=m;   bgc=b;  }
28:         ~Scene(){   for(int i=0; i<n; i++)
29:                         if     (type[i]==WALL)delete (Wall*)obj[i];
30:                         else if(type[i]==BALL)delete (Ball*)obj[i];
31:                         else if(type[i]==CSGM)delete (Csgm*)obj[i];
32:         }
33:         Color   bgcolor( Ray& VR ){ return bgc; }
34:         void    regist(Wall& a, Attri& c)
35:                 {   obj[n]=new Wall; *(Wall*)obj[n]=a; type[n]=WALL; att[n++]=c;    }
36:         void    regist(Ball& a, Attri& c)
37:                 {   obj[n]=new Ball; *(Ball*)obj[n]=a; type[n]=BALL; att[n++]=c;    }
38:         void    regist(Csgm& a, Attri& c, Ball& bv=Ball(Point(0,0,0), -1))
39:                 {   obj[n]=new Csgm; *(Csgm*)obj[n]=Csgm(a,bv); type[n]=CSGM; att[n++]=c; }
40:         Attri   attri(int i, Point& P=Point(0,0,0), Vector& N=Vector(1,0,0) )
41:                 {   return att[i];      }
42:         int     hit  (Ray& VR, Point& P, Vector& N, int& hi);
43:         float   trans(Ray& VR, float l);
44:         Color   color(Ray& VR, Point& L0, int rn){  int r=rn;   return color(VR,L0,r,1);    }
45:         Color   color(Ray& VR, Point& L0, int& rn, float am);
46: };
47:
48: //  視線(反射視線)が物体と交わるかのチェック(交点座標P,物体の番号hiを返す)
49: int     Scene::hit(Ray& VR, Point& P, Vector& N, int& hi){
50:         float   t=INFINITY,tt;                        //  視点から交点までの距離
```

```
 51:         Point   HP,HN;                                      //  交点
 52:         hi=-1;
 53:         for(int i=0,j; i<n; i++){
 54:             if(type[i]==WALL)       tt=((Wall*)obj[i])->hit(VR,HP,HN);
 55:             else if(type[i]==BALL)  tt=((Ball*)obj[i])->hit(VR,HP,HN);
 56:             else if(type[i]==CSGM)  tt=((Csgm*)obj[i])->hit(VR,HP,HN);
 57:             if( DEL<tt && tt<t ){   t=tt;   hi=i;   P=HP;   N=HN;   }
 58:         }
 59:         return hi;
 60: }
 61:
 62: //  照明光の強さを返す
 63: float   Scene::trans(Ray& VR, float l){
 64:         float   s=1,tt;
 65:         Point   HP,HN;                                      //  交点
 66:         for(int i=0; i<n; i++){
 67:             if(type[i]==WALL)       tt=((Wall*)obj[i])->hit(VR,HP,HN);
 68:             else if(type[i]==BALL)  tt=((Ball*)obj[i])->hit(VR,HP,HN);
 69:             else if(type[i]==CSGM)  tt=((Csgm*)obj[i])->hit(VR,HP,HN);
 70:             if( DEL<tt && tt<l-DEL )
 71:                 if(att[i].kn<=0)s=0;    else s=s*(1-att[i].kr);
 72:             if( s<=0) break;
 73:         }
 74:         return s;
 75: }
 76:
 77: //  視線（反射屈折視線）上の物体の色彩,  rn:反射屈折の制限回数
 78: Color   Scene::color(Ray& VR, Point& L0, int& rn, float am){
 79:         if(rn==0)    return BLACK;                           //  反射屈折回数の制限
 80:         if(am<0.01) return BLACK;
 81:         Point   Ps,N;
 82:         int     hi=-1,  rnt=rn-1,   rnr=rn-1;               //  hi:  視線が交わった物体の番号
 83:         if(hit(VR,Ps,N,hi)<0)
 84:             {   rn=0;   return bgcolor(VR); }
 85:
 86:         Color   Cs,Cr,Ct=BLACK;
 87:         Attri   A=attri(hi,Ps,N);
 88:         float   kr=A.kr;
 89:         Vector  L =unit(Ps-L0);                             //  照明光方向ベクトル
 90:         Ray     LR=Ray(Ps,-L);                              //  照明光線の記述
 91:         float   s=trans(LR,len(L0-Ps));                     //  交点が日陰
 92:         Cs=shading(VR.d,L,N,A,s);                           //  交点自身の色
 93:
 94:         if(A.kn>0){                                         //  屈折光の処理
 95:             float   n=A.kn;
 96:             if(N*VR.d>0){   n=1/n;  N=-N;   }               //  物体から出る場合
 97:             float   c=-VR.d*N,  g2=n*n+c*c-1;
 98:             if(g2>0){                                       //  全反射でない
 99:                 float   g=sqrt(g2);
100:                 Ray     TR=Ray(Ps,(VR.d+(c-g)*N)/n);
101:                 kr=((c-g)*(c-g)/(c+g)/(c+g)
102:                     +(n*n*c-g)*(n*n*c-g)/(n*n*c+g)/(n*n*c+g))/2;
103:                 Ct=color(TR,L0,rnt,am*(1-kr)*A.kt);
104:             }   else kr=1;                                  //  完全反射
105:         }
106:
107:         if(kr>0){                                           //  屈折光の処理
108:             if(N*VR.d>0)N=-N;
109:             Ray     RR=Ray(Ps,VR.d-2*(VR.d*N)*N);           //  反射視線の記述
110:             Cr=color(RR,L0,rnr,am*kr);                      //  反射視点から見える物体の色
111:         }
112:         rn=max(rnr,rnt)+1;
113:         return  kr*Cr+(1-kr)*A.kt*Ct+Cs;
114: }
115: #endif
```

**著 者 略 歴**

小笠原祐治（おがさわら・ゆうじ）
1981年　岩手大学工学部情報工学科卒業
1983年　岩手大学大学院情報工学専攻修了
1983年　富士通㈱入社
1991年　岩手大学工学部情報工学科助手
1997年　岩手県立産業技術短期大学校情報技術科講師
現在に至る

C＋＋による簡単実習
3次元CG入門


1999年4月15日


著　者　小笠原祐治
発行者　森北　肇
発行所　**森北出版株式会社**
東京都千代田区富士見1-4-11（〒102-0071）
電話　03-3265-8341／FAX 03-3264-8709
http://www.morikita.co.jp/
自然科学書協会・工学書協会　会員
R＜日本複写権センター委託出版物・特別扱い＞

落丁・乱丁本はお取替え致します　印刷/中央印刷・製本/石毛製本

Printed in Japan／ISBN4-627-84141-8